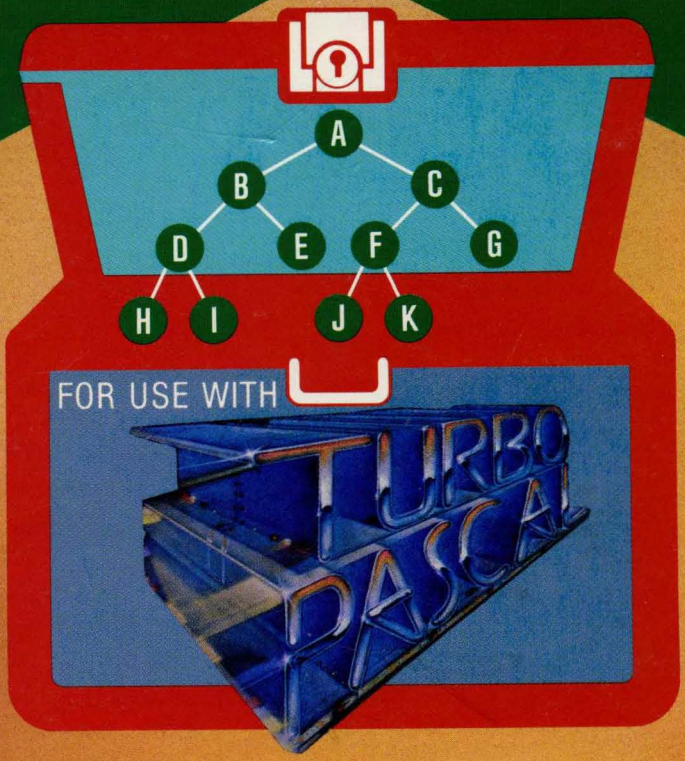
# TURBO TOOLBOX™

## POWER TOOLS FOR TURBO PASCAL

- TURBO-ISAM Files Using B+ Trees
- Quicksort on Disk
- GINST (General Installation Program)




BORLAND INTERNATIONAL

Borland International
4113 Scotts Valley Drive
Scotts Valley, California 95066

# TURBO TOOLBOX™

*Version 1.0*

## Reference Manual

## Copyright Notice©



## Single CPU License

The price paid for one copy of TURBO TOOLBOX licenses you to use the product on one CPU when and only when you have signed and returned the License Agreement printed in this book. In addition, you are allowed to include TURBO Access and TURBO Sort object code (only) modules in programs for general sale and/or distribution.

## Disclaimer

Borland International makes no warranties as to the contents of this manual and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Borland International further reserves the right to make changes to the specifications of the program and contents of the manual without obligation to notify any person or organization of such changes.

This manual is the result of a joint effort between Borland International of Scotts Valley, California, and Technology Writing Services of San Jose, California.

First edition, July 1984
Printed in the United States of America
9 8 7 6 5 4 3 2 1

RETURN THIS LICENSE AGREEMENT TO BORLAND INTERNATIONAL

**Program License Agreement**

On the condition that you sign and return this **License Agreement** to Borland International Inc., Borland hereby grants you a non-exclusive and nontransferable license to use the copy of the software TURBO Toolbox as specified in this agreement on one CPU only, with the additional understanding that **object code only** resulting from inclusion of TURBO Access and/or TURBO Sort source code in your TURBO Pascal programs may be distributed, given away, or sold as part of your programs. Failure to sign this agreement and still use the software is illegal.

I agree that TURBO Toolbox remains the property of Borland International, Inc. and accept that giving away or selling copies of TURBO Pascal is theft of Borland's property and understand that this will be prosecuted by the lawyers of Borland International.

The reference manual is, of course, my property but copyrighted by Borland International Inc. as printed in the manual.

Borland warrants that all material furnished by Borland constitutes an accurate original manufacture of TURBO Toolbox and will replace any such material proven to be defective, provided that such defect is found and reported *within ten days after purchase.* Borland makes no other express or implied warranties with regard to performance or accuracy of TURBO Toolbox and pertaining to documentation and specifically disclaims any implied warranties of fitness for any particular purpose. I agree that Borland shall not be held responsible for any consequential damages that I may possibly incur through the use of TURBO Toolbox, whether through Borland's negligence or not.

**Termination of License.** Any breach of one or more of the provisions of this agreement by me shall constitute an immediate termination of this agreement. Nevertheless, I agree that in the event of such termination, all provisions of this agreement which protect the rights of **Borland** shall remain in force.

**I hereby acknowledge that I have read this agreement, understand it, and agree to be bound by its terms and conditions.**

Name and signature: _____

Address: _____

_____ My serial no: _____

USER'S COPY

**Program License Agreement**

On the condition that you sign and return this **License Agreement** to Borland International Inc., Borland hereby grants you a non-exclusive and nontransferable license to use the copy of the software TURBO Toolbox as specified in this agreement on one CPU only, with the additional understanding that **object code only** resulting from inclusion of TURBO Access and/or TURBO Sort source code in your TURBO Pascal programs may be distributed, given away, or sold as part of your programs. Failure to sign this agreement and still use the software is illegal.

I agree that TURBO Toolbox remains the property of Borland International, Inc. and accept that giving away or selling copies of TURBO Pascal is theft of Borland's property and understand that this will be prosecuted by the lawyers of Borland International.

The reference manual is, of course, my property but copyrighted by Borland International Inc. as printed in the manual.

Borland warrants that all material furnished by Borland constitutes an accurate original manufacture of TURBO Toolbox and will replace any such material proven to be defective, provided that such defect is found and reported *within ten days after purchase.* Borland makes no other express or implied warranties with regard to performance or accuracy of TURBO Toolbox and pertaining to documentation and specifically disclaims any implied warranties of fitness for any particular purpose. I agree that Borland shall not be held responsible for any consequential damages that I may possibly incur through the use of TURBO Toolbox, whether through Borland's negligence or not.

**Termination of License**. Any breach of one or more of the provisions of this agreement by me shall constitute an immediate termination of this agreement. Nevertheless, I agree that in the event of such termination, all provisions of this agreement which protect the rights of **Borland** shall remain in force.

**I hereby acknowledge that I have read this agreement, understand it, and agree to be bound by its terms and conditions.**

Name and signature: _____

Address: _____

_____ My serial no: _____

# TABLE OF CONTENTS

# APPENDIX



# LIST OF ILLUSTRATIONS

# INTRODUCTION

This book is a reference manual for the TURBO TOOLBOX as implemented for the **CP/M-80**, **CP/M-86**, and **MS-DOS** operating systems. Although making thorough use of examples, this manual does not explain how to program in Pascal nor how to use TURBO Pascal. A fairly thorough knowledge of Pascal and some amount of actual TURBO Pascal programming experience are assumed. If you have not already done so, you should read your copy of the *TURBO Pascal Reference Manual* and familiarize yourself with this implementation of Pascal.

## The TURBO Toolbox

The TURBO Toolbox is a set of programs ("tools") which you use with TURBO Pascal to help you in program development. The Toolbox contains three important tools:

- TURBO Access system
- The TURBO Sort system
- The GINST general installation system

These three tools are provided in such a way that they can be included in your Pascal programs for the benefit of your users. In this way, you save hours of development time while making your programs easy to use.

## The TURBO Access System

The TURBO Access system makes it possible to access records in a file by using *keys* (for example, SMITH or JANUARY 21) instead of by number only. In addition to accessing files by key, you can also access them in a sorted sequence. This is a very important tool if you are working with direct access files.

Using the TURBO Access system, files are accessed via B-trees. B-trees are briefly explained in this manual, and references are provided to other books for a more complete discussion.

This tool is provided in such a way that you can provide its capabilities to your users without doing extensive programming.

## The TURBO Sort System

Sorting is either a mystery or a problem area for many programmers. The TURBO Sort system is the part of the TURBO Toolbox which answers this need.

The TURBO Sort system uses the Quicksort method to ensure fast and efficient sorting. With this tool, you can sort any type of data either on a single item or on multiple keys. You can also sort different data items in the same program.

This tool is also provided so that you can provide it to your users and save valuable development time.

## GINST—The General Installation System

GINST solves another potential problem facing programmers: how to make your programs usable on various terminals. GINST is a program that lets you develop an installation module identical to TURBO Pascal's.This allows your users to install your programs for their particular terminals.

The resulting installation program is intended to be included with your programs for use by your customers.

## Structure of This Manual

This manual is divided into three Parts:

- Part I explains the TURBO Access system
- Part II explains the TURBO Sort system
- Part III explains the general installation (GINST) program

Each Part of the manual is further divided into chapters which break the subject matter into "chunks" of a reasonable size. You may read the Part(s) of the manual which pertains to your needs. Each provides a complete description of how the particular tool is used.

## Typography

The body of this manual is printed in normal typeface. Special characters are used for the following special purposes:

Alternate — Alternate characters are used to illustrate program examples and screen display. Screen images are furthermore shown in rectangular fields of thin lines.

*Italics* — Italics are used in general to emphasize sections of the text. In particular, predefined standard identifiers are printed in italics, and elements in syntax descriptions are printed in italics. The meaning and use of italics thus depends upon the context.

**Boldface** — Boldface is used to mark reserved words, in the text as well as in program examples.

You should refer to the *TURBO Pascal Reference Manual* for a complete description of the syntax, special characters, and overall appearance of the Pascal language.

## Copying the Distribution Diskette

The TURBO Toolbox distribution diskette contains several files related to each of the tools. The actual file names belonging to each tool are provided in each Part of the manual, respectively.

The distribution diskette is your only source of the toolbox files. The first thing you should do upon receiving the diskette is to complete and mail the License Agreement at the front of this manual. The second thing you should do is make a copy of the distribution diskette. Then, put the distribution diskette away in a safe place and use the copy for doing your work. If something happens to the copy, make a new copy from the original distribution diskette.

You should never use the distribution diskette for actual work.

## Acknowledgements

In this manual, references are made to several programs. The following provides credit to the owners of these programs:

- SideKick — a registered trademark of Borland International Inc
- TURBO Pascal — a registered trademark of Borland International Inc
- WordStar — a registered trademark of MicroPro International Corp
- CP/M — a registered trademark of Digital Research Inc
- MS-DOS — a trademark of Microsoft Corporation

# PART I

# The TURBO Access System

# 1. INTRODUCTION TO TURBO ACCESS

This is the part of the manual for the TURBO Access system. TURBO Access is used to include fast and efficient access and administration of large data files into TURBO Pascal programs. TURBO Access is a set of source code files for inclusion in your TURBO Pascal programs prior to compilation. Because this system is to be used with TURBO Pascal, the resulting programs can be used with the operating systems under which TURBO Pascal will run. These are CP/M-80, CP/M-86, and MS-DOS. We have assumed that the TURBO Access user already has a working knowledge of the TURBO Pascal language and its fundamental concepts, as they are described and defined in the *TURBO Pascal Reference Manual.*

The source code of TURBO Access can be inserted into programs and compiled with TURBO Pascal Version 2.0 (or more recent versions). If your TURBO Pascal is older than Version 2.0, contact BORLAND INTERNATIONAL for an update.

## 1.1 TURBO Access

The problem of data access is fundamental in computer usage. Even though automatic data processing is fast when compared to manual methods, the time required to store and retrieve the data soon becomes significant when large amounts of data are involved. Among the many methods developed for fast data access, TURBO Access represents one of the most versatile and efficient.

TURBO Access is based upon the B-tree principle, named after R. Bayer, one of its inventors. In a B-tree, a data unit is accessed by using a *key.* Any given key is related to one and only one data unit in a data file. As an example, suitable keys for a file of customer information might be customer names. In other situations, special codes may be more appropriate as keys than customer names. Keys to a file of spare part information could be string representations of the spare part numbers.

In the B-tree, keys are organized in such a way that a search for a particular key demands very few accesses of background storage. At the same time, it is still possible to access the keys in alphabetical order.

**With TURBO Access it is possible to maintain several data files and to access each of these files with several keys. Different data files can also be accessed by using the same key.**

In most cases, the data units to be stored or retrieved take up much more space than their keys. This slows down the search process and is one of the reasons why the relationship between keys and data units are often stored in separate, special files.

A special file of keys makes it possible to use unchanged general B-tree procedures in any particular context. Throughout this manual we will call these special files *Index files*. The files containing data units to be stored or retrieved, on the other hand, are called *Data files*. Because a record in an *Index file* contains little more than one key and one reference to the data unit in a *Data file*, its transfer from background storage is very quick. In this way, only one access is made to the *Data file* for each data unit to be accessed. If the key is not found in the *Index file*, the *Data file* is not accessed at all (except in the situation where the next step is to be the insertion of a new data unit into the *Data file*.

As already mentioned, the organization of data references as a B-tree has no consequences whatsoever with respect to the information in the *Data file*. Administration of the *Data file* is exclusively the responsibility of the user. Although this administration is facilitated by the TURBO Access procedures, the programmer must still determine how the *Index files* will be used on the one hand, and how the *Data files* will be used on the other hand.

## 1.2 TURBO Access Procedures

High level computer languages all offer the facility of procedures in programs. One advantage of procedures is that once they are defined, their internal structures are detached from the uses to which they will be put. To use procedures in a program it is necessary only to know what they do under various circumstances. The standard procedures "Read" and "Write" are good examples of this.

There are several possible actions involved in the management of *Index files* organized as B-trees. For example, searching for a particular key, and inserting or deleting a key along with its associated data item are trivial examples; however, the time and effort required to program these actions is by no means trivial. With TURBO Access, the program code for *Index file* management is supplied as procedures to be included directly into the user's program. This makes it possible for you to enjoy the advantages of fast and reliable data access without programming a B-tree based system yourself. It is not even necessary to understand what B-trees are all about. It must be emphasized, however, that the use of TURBO Access procedures will be more meaningful when you understand some basic facts about B-trees.

# 2. FILE ACCESS USING B-TREES

This chapter explains B-tree structure and management. It may serve as a general introduction to the subject of B-trees with the exception that those aspects of B-trees which do not relate to the TURBO Access implementation are not covered. A thorough treatment of B-trees can be found in the following books:

- Wirth, Niklaus: *Algorithms + Data Structures = Programs.* Prentice Hall (1976).
- Knuth, Donald E.: *The Art of Computer Programming* (Vol. 3). Addison Wesley (1973)
- Horowitz, E. et al: *Fundamentals of Data Structures.* Pitman (1976)

This chapter covers only subjects and structures relevant to the functional principles of B-trees.

## 2.1 Standard Data Access

The fundamental way of storing data is to organize it into similar units. These units can be of any type, standard as well as user-defined (except files). For complex data, the usual choice for a unit is a user-defined record. The units are then arranged into a file by using a file declaration in the program. (For details on the declaration and use of files, refer to the *TURBO Pascal Reference Manual*: **File Types**). In TURBO Pascal there are two ways to access the units of a file:

- Sequential data access
- Random data access

Each of these methods is discussed in the following subsections.

### 2.1.1 Sequential Data Access

The simplest way to find a particular unit of data is to search through the file starting from the beginning. The following is an example of code to accomplish this:

```
.
PresentUnit, WantedUnit          : Unit;
DataFile                         : File of Unit;
.
.
Reset (DataFile);
Repeat
Read(DataFile,PresentUnit);
Until WantedUnit = PresentUnit;
.
```

It is obvious that this method becomes increasingly slow as the *Data file* grows. Every unit prior to the desired unit must be completely examined before the search can be completed.

### 2.1.2 Random Data Access

By using random access techniques, it is possible to avoid unnecessary reading of the data file. Random access is performed by using the actual number of the desired unit of data. In this method, the first unit of data stored is number 0 (zero). This number is called a *data reference*. The process has two steps:first, the file pointer is positioned at the data unit to be read, and second, the read operation is performed. The following example shows how this type of operation is coded:

```
.
DataRef                          : 0..maxint;
PresentUnit, WantedUnit          : Unit;
DataFile                         : File of Unit;
.
.
Seek(DataFile,DataRef);
Read(DataFile,PresentUnit);
.
```

Contrary to what is implied by the term, random access demands that the user knows the data reference to each unit in a *Data file*.

In TURBO Access, this vital information is supplied by the B-tree procedures working on the *Index files*. As we shall see, an *Index file* is itself organized with units that contain references to other units in the same file.

## 2.2 Structure of B-Trees

This section provides information about the structure of B-trees.

### *2.2.1 Keys*

As mentioned in Chapter 1, *key* manipulation in the TURBO Access system does not directly affect the *Data files*. The user handles the *Data files* with the aid of *data references* stored in *Index files*.

References for units of a *Data file* are stored and retrieved by means of *keys*. The *keys* are strings of a user-specifed maximum length (less than or equal to 255 characters). The construction of the key for a particular data object is exemplified in the sample program of Chapter 7. In this example, the first and last names of persons are merged to form a key with a maximum length of 25 characters.

**Example:**

John Jones → 'Jones John'

Strings use an ordering system which uses the relational operators "=", "<", and ">". This ordering is described in the *TURBO Pascal Reference Manual:* **String type**. It follows that there is only one way to place any set of different keys in order. The B-tree principle is based upon the assumption that no two *data references* have the same key. Consequently, TURBO Access procedures will make certain that there are no *duplicate keys in the Index file* before the user is allowed to insert a new key. However, there is a way to handle duplicate keys if this is specified when creating an *Index file*. The way in which duplicate keys are handled is not included in the following description of B-tree principles.

The Random Access Memory (RAM) space required by the working B-tree procedures grows with increasing key lengths. The key length must allow for a sufficient number of possibilities of variations to separate all entries in the *Index file*. In many cases, abbreviations of the original key information will serve the purpose.

**Example:**

Henry Smidth, Plumber → 'PlumSmidH'

If the program uses file access in key order, it is advisable to use only either upper or lower-case letters for keys, because the ordering is based upon the ASCII character value (See Appendix B) of the string components. If no measures are taken to avoid problems caused by the wrong case, this may produce unexpected sequences of keys.

**Example:**

'RHINOCEROS' < 'mouse'

If it is appropriate to use a number as key, the number must be transformed into a string representation. A simple way is to use the *Str* procedure (refer to the *TURBO Pascal Reference Manual:* **String Procedures**). A more sophisticated method will be covered later.

**Example:**

```
.
var SerialNumber,
Code                    : integer;
Key                     : String[6];
.
.
Key := Str(Key,SerialNumber,Code);
.
```

## 2.2.2 *Items*

The connection between a key and its reference to a *Data file* is formalized as a **record** called an item (see Figure 2-1). In addition to these two components, the item holds an internal *Index file* reference (*Page reference*), which is used in the building of the tree structure.

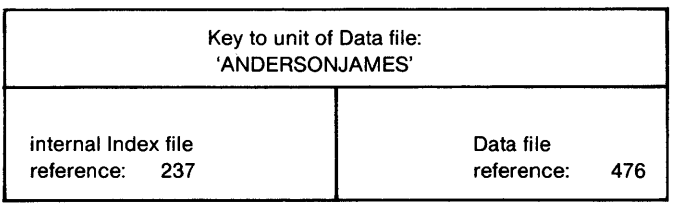



Figure 2-1: Structure of an Item in the *Index File*

### 2.2.3 Pages

The units of an *Index file* are called pages (see Figure 2-2). On each page is found an even number of *items*. The first page is called the *root page*. All other pages are always at least half full; that is, at least half of their items hold a key, a page reference, and a data reference. This is why the *order* (**n**) of the B-tree is defined as half of the maximum item number per page. In addition to the items, the page contains an extra page reference not connected to any item.

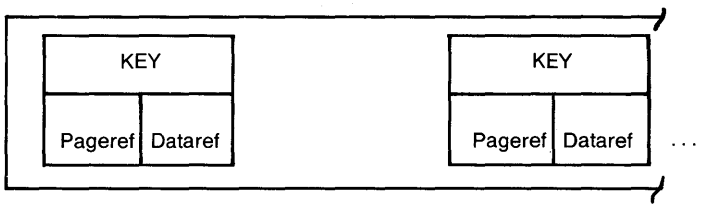



Figure 2-2: Structure of a Page in the *Index File*

### 2.2.4 Pages into Trees

As we have seen, each key in a page item holds a possible reference to another page in the data structure. In a B-tree, the page to which the page reference points contains additional keys. All of these are larger than the key associated with the page reference itself. Thus, consecutive jumps from a page item to the page of its reference will create a *path* of ever-increasing keys. The path stops at the terminal page with no active page references. The terminal pages are called leaf pages.

It is also possible to move from one page to another in such a way that any key in the second page is smaller than any key in the first page. This is done by using the extra page reference held by all pages. The single path made from only backward references will end at the leaf page with the smallest keys in the B-tree.

All paths in the B-tree start at the root page, and each jump can be to either larger or smaller keys. The ordering of keys in a B-tree has the consequence that each page in the tree can be reached by one and only one path. The number of jumps in this path is referred to as the *level* of the page. (The root page is at level 1.)

### 2.2.5 *Finding the Data Reference*

The purpose of a B-tree is to provide a reference to a *Data file* by specifying a key. The page with the data reference is found by following its path starting at the root page. This is a guided tour, because comparison with the keys on pages encountered along the way give exact information on the next jump. If all keys on the page are larger than the key you are seeking, the next page to be investigated is referenced by the backward page reference. If this is not the case, the reference to the next page is found in the item with the largest key which is less than the key you are seeking. The search continues until the key is found or until a leaf page is reached with no result.

### 2.2.6 *Formal Properties of B-trees*

The structure of B-trees is subject to several constraints that serve to keep the search efficiency intact under all circumstances of key insertion and deletion. The constraints (some of which have already been mentioned) are as follows:

- All pages hold at most **2n** items (keys)
- With the exception of the root page, all pages hold at least **n** items (keys)
- Every page is either a leaf page with no active page references or it has **m** + 1 active page references (where **m** is the number of keys on the page (**n** <= **m** <= **2n**)).
- All leaf pages are at the same level (called the tree height).

The advantage gained by adhering to these rules are several. They ensure that all parts of the B-tree have a minimum information density. At the same time, most of the keys have the same search path length; that is, the current tree height is the same. The remainder have shorter paths.

Each time the B-tree height grows by one due to insertion of new keys, the room for additional keys is increased by a factor of between (**n** + 1) and (**2n** + 1) compared to the previous level. The maximum number of keys in the TURBO Access system is MaxInt. Given the *order* 3, this number of keys can be searched with a maximum path length of 7, and in most cases less. If we choose an *order* of 16, the corresponding number is 4. Without these restrictions in the B-tree structure, it would be impossible to predict bounds for the path length of a tree. If this were true, path length would depend upon the history of key insertion.

## 2.3 B-tree Administration

The rules given in Section 2.2.6 necessitate regular rearrangement of the page items in the B-tree, when keys are inserted or deleted. This is all taken care of by the TURBO Access procedures. We will present simple examples of this here, but will refer you to the literature for details.

### *2.3.1 Inserting Keys*

When a key is presented to the B-tree for insertion, a check is first performed to determine whether or not the key is already present in the tree. If the key is new, the search stops at a *leaf page* with a negative result. This leaf is the appropriate place for insertion of the new key. The only problem that can occur is when the leaf page is full (that it, it already contains 2**n** items).

This problem is solved by page splitting. The (2**n** + 1) keys are redistributed onto two pages, one of which is new. The n largest keys are moved to a new leaf page, and key number (**n** + 1) is moved to the ancestral page of the leaf, where it is associated with a reference to the new leaf. This preserves the ordering of the keys affected by the operation.

If the ancestral page is full, it must be split to accept the key moved from the leaf level. In this way, page splitting may propagate from the leaves all the way to the root. When that happens, splitting of the root results in creation of a new root, and the B-tree height is increased by one. Immediately after this occurs, the new root holds only one item—the one moved from the previous root. The B-tree grows in height by addition of a new root—a simple way to ensure the requirement that all leaves must be at the same level.

The possibility of propagating page splitting makes it necessary to keep all pages of the search path in RAM (called the *B-tree stack*) during the insert procedure. In addition to this, the stack must also have room for a new page. TURBO Access makes certain in advance that this space will be available.

### *2.3.2 Deleting Keys*

When deletion of an existing key is requested, its location in the tree is found. If this turns out to be on a leaf page, removal is simple. If the key is situated elsewhere in the tree, the process of deletion is more complicated.

The vacant page reference must be connected to another key in the tree, without destroying the key *order*. This must be a key at leaf level, as all other keys are engaged by page references already. Fortunately, it is always possible to find such a key. The smallest key which is larger than the one to be deleted is always on a leaf page. Likewise, the largest key which is smaller than the one to be deleted is also on a leaf page. One of these can be used without affecting the key ordering.

Moving a key from leaf level to take over the page referencing job of the deleted key only solves half of the problem. The leaf page, from where the key came, may now have too few items (that is, less than **n**). If one of the adjacent leaves has more than n items, it is possible to redistribute items so that the two leaf pages have a legal number of items. Preservation of key *order* makes it necessary to shift items via the ancestral item separating the leaves in question. This process is called *balancing*.

If no adjacent leaf page has a surplus of items, then it is possible to *merge* the "underflow leaf" with one of them. This process is exactly the reverse of *page splitting*, which was described in Section 2.3.1. Like page splitting, merging of pages may propagate to lower levels of the tree. In the extreme case, B-tree height is reduced by one. It is necessary to have space for all pages of the search path plus 1 on the *Heap* at key deletion. Space availability is determined beforehand by the TURBO Access procedure.

# 3. THE TURBO ACCESS SYSTEM

This chapter provides a functional description of the TURBO Access system.

## 3.1 Main Components

TURBO Access is supplied as four TURBO Pascal source files. These TURBO Access modules are included in the compilation of an application program through the {$I filename} directive of the TURBO compiler. The TURBO Access files are:

- **ACCESS.BOX** — Contains the basic *Data/Index file* setup and data file maintenance routines. This module must always be included before any other TURBO Access modules.
- **GETKEY.BOX** — Contains the search routines of TURBO Access. The components are the *NextKey, PrevKey, FindKey,* and *SearchKey* procedures.
- **ADDKEY.BOX** — Contains the routine used for inserting keys into *Index files.* This is done with the *AddKey* procedure.
- **DELKEY.BOX** — Contains the routine used to delete keys from *Index files.* This is done with the *DeleteKey* procedure.

The ACCESS.BOX module must always be included in a program that uses TURBO Access and it must always be the first module. GETKEY.BOX, ADDKEY.BOX, and DELKEY.BOX, however, may be included or omitted as required, and their order is of no importance. As long as the precedence of ACCESS.BOX to the other modules is observed, it is also possible to place the modules in program overlays.

## 3.2 Function Summary

TURBO Access routines fall into four categories:

- *Data file* initialization and update routines
- *Index file* initialization routines

- *Index file* update routines
- *Index file* search routines

The functions of each of these categories are summarized in the following subsections. The descriptions also specify which TURBO Access module must be included in your program to achieve the function.

### 3.2.1 Data File Initialization and Update

| | |
|---|---|
| *MakeFile* | Creates a *Data file* (DTAMAN). |
| *OpenFile* | Opens an existing *Data file* (DTAMAN). |
| *CloseFile* | Closes a *Data file* (DTAMAN). |
| *AddRec* | Allocates a new data record. *AddRec* automatically reuses previously deleted data records before extending the *Data file* (DTAMAN). |
| *DeleteRec* | Deletes a data record. The record is entered in the list of deleted records, and will be reused by *AddRec* before the *Data file* is extended (DTAMAN). |
| *GetRec* | Reads a data record (DTAMAN). |
| *PutRec* | Writes a data record (DTAMAN). |
| *FileLen* | Returns the total number of records in a *Data file*, including deleted and reserved records (DTAMAN). |
| *UsedRecs* | Returns the number of data records in use in a *Data file* (DTAMAN). |

### 3.2.2 Index File Initialization

| | |
|---|---|
| *InitIndex* | Initializes the TURBO Access *Index file* buffers (DTAMAN). |
| *MakeIndex* | Creates an *Index file* (DTAMAN). |
| *OpenIndex* | Opens an existing *Index file* (DTAMAN). |
| *CloseIndex* | Closes an *Index file* (DTAMAN). |

### 3.2.3 Index File Update

*AddKey* — Adds a new key string and its associated data record number to an *Index file* (ADDKEY).

*DeleteKey* — Deletes a key string and its associated data record number from an *Index file* (DELKEY).

### 3.2.4 Index File Search

*FindKey* — Finds an *Index file* entry which is equal to a given search string (KEYLOC).

*SearchKey* — Finds the first *Index file* entry which is equal to or greater than a given search string (KEYLOC).

*NextKey* — Returns the *Index file* which immediately follows the last entry located by a search routine (KEYLOC).

*PrevKey* — Returns the index entry which immediately precedes the last entry located by a search routine (KEYLOC).

*ClearKey* — Sets the index pointer to point to the beginning or end of an *Index file* for sequential processing (DTAMAN).

## 3.3 Types and Variables

In addition to the basic *Data/Index file* maintenance routines, the TURBO Access DTAMAN module also defines some global types and variables. They are:

*DataFile* — This type of identifier is used to declare the *Data file* variables. All TURBO Access *Data files* are declared with this identifier, even though their data records are not of the same type and size.

*IndexFile* — This type of identifier is used to declare *Index file* variables.

*OK* — A boolean variable used to return the status of some TURBO Access routines.

## 3.4 General Notes

TURBO Access *Data files* may contain up to 65536 records and the record size, in theory, may be up to 64K bytes.

In contrast to ordinary TURBO Pascal *Data files*, TURBO Access *Data file* variables are always declared using the DataFile type (the actual record size is specified at run time when a new *Data file* is created).

TURBO Access *Data files* always have fixed size records. The minimum data record length is 8 bytes.

The first record of a *Data file* (record number 0) is reserved for system use (the *AddRec* routine will automatically reserve this record).

TURBO Access makes not assumptions about the relationship between *Index files* and *Data files*. It is the responsibility of your application program to extract key information from each data record and enter it into an *Index file* along with a data record number. As a consequence, several *Index files* may reference one *Data file*. It is recommended that you included the key values both in the *Data file* records and in the *Index files* since this will greatly simplify the task of reconstructing a corrupted index. It also enables you to use computed index keys (that is, keys which are computed from one or more fields in a data record).

For example, suppose your application program is designed to maintain a mailing list. The keys stored in the index might then be the last name of each entry in the *Data file*, converted to upper case. If a search string is also converted to upper case, TURBO Access will find the appropriate entry regardless of whether it was entered using upper- and/or lower-case letters, but the data record will still contain the data exactly as entered. In general, key values should only be omitted from the *Data file* records if you have a very limited amount of disk storage.

## 3.5 Using TURBO Access

As stated earlier, the TURBO Access routines are made available to your application program by including one or more of the four TURBO Access modules. The DTAMAN module must always be included before other TURBO Access modules since it contains the basic disk file interface. Following DTAMAN, you may then include the KEYLOG, ADDKEY, and DELKEY modules as required.

Prior to including DTAMAN, some integer constants must be declared:

*MaxDataRecSize* Maximum record length. *MaxDataRecSize* should be set to the size (in bytes) of the largest data records your program is going to process (that is, if your program will process two *Data files* with record sizes of 72 and 140 bytes, then *MaxDataRecSize* should be set to 140).

*MaxKeyLen* — Maximum key length (an integer between 1 and 255). *MaxKeyLen* should be set to the largest maximum key length of the *Index file* your program is going to process (that is, if your program will process three *Index files* with maximum key lengths of 16, 10, and 25, then *MaxKeyLen* should be set to 25).

*PageSize* — The maximum number of key entries allowed in each *Index file* record (page). The page size must be the same for all *Index files* to be processed by your program. *PageSize* should be an even number between 4 and 254. For further details, see below.

*Order* — Half of *PageSize*. The minimum number of items permissible on an *Index file* page (except the root page). See Section 2.2.6.

*PageStackSize* — Page buffer size. The number of *Index file* records (pages) that can be kept within memory at one time. The minimum value is 3. In general, increasing *PageStackSize* will speed up the system, because the probability that an *Index file* record is already within memory when it is to be processed increases.

*MaxHeight* — The maximum height of the *Index file* B-tree structures. This constant applies to all *Index files* to be processed by your program. For a calculation of *MaxHeight*, see below.

The maximum number of pages (K) to be searched to find a specific key in an *Index file* with E keys is approximately:

K = Log(E) / Log(*PageSize* * 0.5)

Thus, a search for large pages requires fewer disk accesses, and therefore less time, than for small pages. The time required to perform a search within the page, once it has been read into memory, is of no significance compared to the time it takes to read the page from the disk. The *MaxHeight* parameter required by DTAMAN corresponds to the integer part of (K + 1); therefore, once you have established the page size and the maximum number of data records in your data base, you compute *MaxHeight* from:

*MaxHeight* = Int (Log(E) / Log(*PageSize* * 0.5)) + 1

Note that larger values of *MaxHeight* will only require very little extra memory. It is therefore recommended that you add 2 or 3 instead of 1, to be on the safe side.

The number of bytes (N) occupied by each page in an *Index file* is found as follows, where KeySize is the key length for that specific *Index file*:

N = (KeySize + 5) * *PageSize* + 3

The maximum number of bytes (D) occupied by an *Index file* is:

D = N * E / (*PageSize* * 0.5)

This formula for D is based on a worst-case assumption, where all pages are assumed to be only half full. Experience shows that the multiplication factor for *PageSize* to be around 0.75.

The number of bytes (M) occupied in memory by the TURBO Access page buffer is:

$$M = ((MaxKeyLen + 5) * PageSize + 3) * PageStackSize$$

where *MaxKeyLen* is the largest key length used by the *Index file* to be processed, and *PageStackSize* is the maximum number of pages that may be kept within memory at the same time (note that this must be at least 3).

It is difficult to devise a general method for calculating the optimum values of *PageSize* and *PageStackSize. PageSize* usually lies between 16 and 32, depending on the maximum key size and the number of keys in the index. Smaller values will result in poor performance due to the time required for key search, and larger values will require too much memory for page buffers.

The minimum reasonable value for *PageStackSize* is the value of *MaxHeight.* If *PageStackSize* is less than *MaxHeight*, the same page will need to be read several times to traverse the B-tree sequentially. In general, *PageStackSize* should be as large as possible. Specifically, if *PageStackSize* is much larger than *MaxHeight*, TURBO Access may keep the root page and the entire first level of the B-tree within memory, thus reducing by at least one the disk accesses required to look up a key.

## 3.6 Skeleton Program

A skeleton program which uses TURBO Access might look like this:

```
program YourProg;

const
  MaxDataRecSize   = 132;    {Maximum record size    }
  MaxKeyLen        =  25;    {Maximum key length     }
  PageSize         =  24;
  Order            =  12;    {PageSize / 2           }
  PageStackSize    =   8;    {Page buffer size       }
  MaxHeight        =   5;    {Maximum B-tree height  }

{$I ACCESS.BOX}

  {Include other TURBO Access modules}

  {Your declarations}

begin

  {Your main program}

end.
```

## 3.7 Program Structure with TURBO Access

In most cases, an application program will do one or more of the following four functions:

- Add data records
- Retrieve data records
- Update data records
- Delete data records

Also, prior to processing any data, an application program must prepare (open) the necessary *Index* and *Data files*, and at termination, close these files.

### *3.7.1 Initializing the User Program*

The initialization phase consists of calling either *MakeFile* (for a new file) or *OpenFile* (for an existing file) for each *Data file* to be used, and *MakeFile* (new) or *OpenIndex* (existing) for each *Index file*. In addition, *InitIndex* must be called to initialize the *Index file* manager routines (this need only be done once, for example at the very beginning of the application program, and only if any of the index routines are to be used).

### *3.7.2 Adding Data Records*

To add a data record, first input the record, and then add it to the *Data file* by using *AddRec*. Then, compute a key value for the new record (this may simply be a field in the data record), and call *AddKey* to enter it into the *Index file*. When you call *AddKey*, you must give the data reference returned by *AddRec*, so that it may be entered into the index along with the key. If your program maintains more than one index, call *AddKey* for each *Index file*, passing over the same data reference each time.

### *3.7.3 Key Location*

To locate a key value, you may use the *FindKey, SearchKey, NextKey, PrevKey,* and *Clearkey* routines to search the index (or indexes). Once the key has been found, you may use *GetRec* to obtain the associated data record from the *Data file*.

### 3.7.4 Key Deletion

To delete a data record, first find its key with *FindKey, SearchKey, NextKey, PrevKey,* or *ClearKey* as just described. Then call *DeleteKey* to delete the record from the *Index file. DeleteKey* will return the data reference connected to the key just deleted. If there is more than one index, then read the data record (*GetRec*), and derive from it the keys to be deleted from the other *Index files.* Finally, call *DeleteRec* to remove the data record from the *Data file.*

### 3.7.5 Key Change

Changes made in a data record may affect the key value(s). For example, the hairdresser 'Andy' may change his name to 'Antoine'. If so, you must call *DeleteKey* to delete the old key and *AddKey* to add the new key. If there is more than on *Index file,* this procedure must be repeated for each file. Finally, call *PutRec* to update the record in the *Data file.*

### 3.7.6 User Program Termination

At termination, your program must call *CloseFile* for each *Data file* in use, and *CloseIndex* for each *Index file.*

### 3.7.7 User Program Variables

The TURBO Access modules contain a number of internal variables. To avoid duplication of their names in your user programs, they all begin with the characters 'TA'. This does not preclude the use of variables starting with 'TA', but can result in compiler error number 43: 'Duplicate identifier or label'. The internal variables found in the source code have been named to promote easy understanding. The actual TURBO Access include files may use other names for internal variables.

# 4. TURBO ACCESS ROUTINES

This chapter describes the procedures made available to the TURBO Pascal programmer by the TURBO Access utility.

Most TURBO Access routines return a status value by using a boolean variable called *OK*, which is automatically declared by TURBO Access. For example, the *OpenFile* procedure sets *OK* to *True* if the file to be opened was found, and sets it to *False* if it was not found. In case of severe or unrecoverable errors, a procedure called *IOcheck* (which is located in the **ACCESS.BOX** module) gains control. *IOcheck* outputs the name of the file, the record number, and the error code, and terminates the program.

## 4.1 Data File Procedures

TURBO Access provides *Data file* support routines for the following tasks:

- Create, open, and close *Data files*
- Add and delete records to and from *Data files*
- Read and write records from and to *Data files*
- Report the size of *Data files*

### *4.1.1 Reuse of Deleted Data Records*

As mentioned earlier, TURBO Access will automatically reuse previously-deleted data records before expanding a *Data file* when new records are added. This is achieved by maintaining a linked list of deleted data records -when a data record is deleted, its first two bytes form a pointer to the next deleted record. Minus one (-1) indicates that the record is the last record in the list. Since a zero pointer (two bytes of zero) never occurs, you may reserve the first two bytes of each data record, and set them to zero when you add a record to the file. This will enable you to distinguish used records from deleted records if you process the file without reference to an index (for example, when you reconstruct a corrupted *Index file*).

### 4.1.2 MakeFile

*MakeFile* creates a new *Data file* and prepares it for processing.

---

**ACCESS.BOX**:

```
procedure MakeFile(var DatF    :    DataFile;
                       FileN   :    String[14];
                       RecLen  :    Integer    );
```

**Parameters**:

*DatF* The *Data file* variable to be prepared for access. *DatF* must be of type *DataFile*.

*FileN* A string expression of up to 14 characters specifying the name of the new disk file.

*RecLen* The record length in bytes. The minimum record length is 8.

On exit, *OK* is *True* if the file was successfully created. If *OK* is *False*, there was not enough space on the disk for a new file.

### 4.1.3 OpenFile

*OpenFile* opens an existing *Data file* and prepares it for processing by the TURBO Access routines.

---

**ACCESS BOX:**

```
procedure OpenFile(var DatF    :    DataFile;
                       FileN   :    String[14];
                       RecLen  :    Integer    );
```

**Parameters:**

*DatF* The *Data file* variable to be prepared to access. *DatF* must be of type *DataFile.*

*FileN* A string expression of up to 14 characters specifying the name of an existing disk file.

*RecLen* The record length in bytes. The record length must be the same as the one used when the file was created.

On exit, *OK* is *True* if the file was found and opened successfully. Otherwise, *OK* is *False.*

### 4.1.4 CloseFile

*CloseFile* closes a *Data file.*

---

**ACCESS.BOX:**

```
procedure CloseFile(var DatF    :    DataFile);
```

**Parameter:**

*DatF* The *Data file* variable to be prepared to access. *DatF* must be of type *DataFile.*

If you make any changes to a *Data file,* always call *CloseFile* for that file before terminating your program. Failing to do so may cause data to be lost, not to mention possible corruption of the *Data file* structure.

### 4.1.5 AddRec

*AddRec* adds a new record to a TURBO Access *Data file.*

---

**ACCESS.BOX:**

```
procedure AddRec(var DatF      :    DataFile;
                 var DataRef   :    Integer;
                 var Buffer                  );
```

**Parameters:**

*DatF* — The *Data file* variable to which a record is added. *DatF* must be of type *DataFile.*

*DataRef* — Data record number. *AddRec* returns the data record number of the newly allocated data record by using this variable parameter. You should pass this data record number to the *AddKey* index routine when you enter a key value for the data record.

*Buffer* — A variable containing the data record to be added. Since *Buffer* is an untyped parameter, *AddRec* will accept any variable in its place. It is up to you to make sure that the variable passed is of the proper type.

Note that *AddRec* does not return a status value. It returns only if the data record was added to the file successfully. If an I/O error occurs, *IOcheck* will gain control and terminate the program. We suggest that you call *UsedRecs* before calling *AddRec* to make sure that there is enough space on the disk.

If any previously deleted records are available, they are automatically reused before the disk file is expanded.

### 4.1.6 DeleteRec

*DeleteRec* removes a data record from a TURBO Access *Data file*.

---

**ACCESS.BOX**:

```
procedure DeleteRec(var DatF     :     DataFile;
                        DataRef  :     Integer      );
```

**Parameters**:

*DatF* The *Data file* variable from which a record is deleted. *DatF* must be of type *DataFile*.

*DataRef* The data record number. If you are maintaining an index, this number should be passed to the *DelKey* index routine when the key value is deleted.

The record is entered into the deleted data record list, so it may be reused by *AddRec* before the *Data file* is expanded.

### 4.1.7 GetRec

*GetRec* reads a specified data record into memory.

---

**ACCESS.BOX**:

```
procedure GetRec(var DatF     :     DataFile;
                     DataRef  :     Integer;
                 var Buffer   :                );
```

**Parameters**:

*DatF* The *Data file* from which the record is read.

*DataRef* The data record number.

*Buffer* A variable to read the data into. Since *Buffer* is an untyped parameter, *GetRec* will accept any variable in its place. It is up to you to make sure that the variable passed is of the proper type.

### 4.1.8 PutRec

*PutRec* writes a data record to a specified position in a Data file.

---

**ACCESS.BOX:**

```
procedure PutRec(var DatF     :     DataFile;
                     DataRef  :     Integer;
                 var Buffer   :                );
```

**Parameters:**

*DatF* The *Data file* to which the record is written

*DataRef* The data record number.

*Buffer* The variable from which the data is written. Since *Buffer* is an untyped parameter, *GetRec* will accept any variable in its place. It is up to you to make sure that the variable passed is of the proper type.

### 4.1.9 FileLen

*FileLen* returns the number of data records allocated to the *Data file* given by *DatF*.

---

**ACCESS.BOX:**

```
function FileLen(var DatF          DataFile):Integer;
```

**Parameter:**

*DatF* *The Data file* from which the number of records is found.

The length returned by *FileLen* includes the reserved record at the beginning of the file (record 0) as well as all deleted records.

### *4.1.10 UsedRecs*

*UsedRecs* returns the number of records in *DatF* that contain valid data.

---

**ACCESS.BOX**:

```
function  UsedRecs(var DatF    :    DataFile) : Integer;
```

**Parameter**:

*DatF* The *Data file* from which the number of records is found.

In contrast to *FileLen*, this function does not include reserved and deleted records.

## 4.2 Index File Support Routines

TurboAccess provides *Index file* support routines for the following tasks:

- Create, open, and close *Index files*
- Add and delete keys to and from *Index files*
- Search for keys in *Index files*
- Read through *Index files* forwards and backwards.

*Index files* are declared by using the *IndexFile* type identifier. Entries in an *Index file* consist of a key string and a data reference. *Key* strings may assume values of type string. The maximum length of the key strings is determined when the *Index file* is opened. If a key string is too long, it will be truncated when it is passed to an *Index file* support routine.

### 4.2.1 Duplicate Keys

There are many applications where key strings are not guaranteed to be unique, such as an index based upon last names. For this reason, duplicate keys may occur. TURBO Access only allows duplicate keys if the *Status* parameter in the call to *MakeIndex* or *OpenIndex* is 1.

When TURBO Access adds duplicate keys to *Index files*, equal keys are ordered by their data references, so that key entries with low references appear first. Normally, this will correspond to the order in which the keys are entered, since new data records are usually added to the end of *Data files*.

The search routines *FindKey* and *SearchKey* will always locate the first key entry; that is, the key entry with the lowest data record number.

To delete a key from an *Index file* with duplicate keys, it is not sufficient simply to specify the key string, since this string may identify several entries. To select a specific entry, you must also specify the data record number. The *DeleteKey* procedure will delete the key entry only if the key string and the data record number match the values found in the *Index file*.

### 4.2.2 Numeric Keys

If your application program requires numeric key values, you must convert these numeric values to strings before passing them to TURBO Access. There are basically two methods of doing this.

The simplest approach is to convert the numeric value to its ASCII string representation using the *Str* standard procedure (refer to the *TURBO Pascal Reference Manual*: **String Procedures**. If you use this method, the resulting strings must be right justified (this is easily accomplished by specifying a field width in the call to Str). The main disadvantage to this method is that the key length must be set to the maximum number of digits that may occur as opposed to the number of bytes required to store the number in its binary format.

The second approach only applies to integers. It takes advantage of the compactness of integers in binary format. The routines shown below may be used to "pack" and "unpack" integers to and from strings. *IntToStr* converts an integer to a string, and *StrToInt* converts a string into an integer. The strings returned by *IntToStr* are two characters long; the strings passed to *StrToInt* must likewise be two characters long.

```
function IntToStr (N : Integer) : Str[2]
begin
   N: = N + $8000;
   IntToStr:=CHR(Hi(N)) + Chr(Lo(N));
end;

function StrToInt(S : Str[2]): Integer;
begin
   StrToInt:=Swap(Ord(S[1]) +Ord(S[2] + $8000; end;
```

The above routines operate on signed integers (-32768 to 32767). If the integers are to be interpreted as unsigned quantities, simply remove the additions of $8000.

### 4.2.3 Data File Splitting

A TURBO Access *Index file* must be contained in a single disk file. *Data files* may, however, be spread over more than one disk file. The only limitation is that the total number of records may not exceed 65536. The splitting of *Data file* is quite simple to implement and best illustrated by an example.

Assume that each *Data file* can hold 10,000 records and that we need to store up to 30,000 records. We will require three *Data files*. When a record is added to the first file, the data record number is entered directly into the *Index file*. However, when records are added to the second and third file, we add 10,000 or 20,000 to the record number. Later, when the *Index file* is read, record numbers may be divided by 10,000 to determine in which files the records reside, and the remainders of the divisions are the actual data record numbers. Note that since the first record number in a *Data file* is 1, you must subtract 1 before dividing and add 1 to the remainder to produce the correct results.

### 4.2.4 InitIndex

*InitIndex* initializes the table used by the *Index file* routines.

---

**ACCESS.BOX**:

```
procedure InitIndex;
```

No parameters

*InitIndex* must be called before other *Index file* routines. Only one call is required, and it is usually placed at the very beginning of the application program.

### 4.2.5 MakeIndex

*MakeIndex* creates a new *Index file* and prepares it for processing.

---

**ACCESS.BOX**:

```
procedure MakeIndex(var IndexF :    IndexFile;
                        FileN  :    String[14];
                        KeyLen,
                        Status :    Integer     );
```

**Parameters**:

*IndexF* The *Index file* variable to be prepared for access. It must be of type *IndexFile*.

*FileN* A string expression of up to 14 characters specifying the disk file name.

*KeyLen* The maximum length of the key strings to be stored in this file.

*Status* 0 indicates that duplicate keys are not allowed, 1 means that duplicate keys may occur.

On exit, the *OK* flag is set to *True* if the file was created successfully. If *OK* is *False*, there was no space on the disk for a new file.

### 4.2.6 OpenIndex

*OpenIndex* opens an existing *Index file* and prepares it for processing.

---

**ACCESS.BOX**:

```
procedure OpenIndex(var IndexF  :  IndexFile;
                        FileN   :  String[14]
                        KeyLen,
                        Status  :  Integer    );
```

**Parameters**:

*IndexF* The *Index file* variable to be prepared for access. It must be of type *IndexFile*.

*FileN* A string expression of up to 14 characters specifying the disk file name.

*KeyLen* The maximum length of the key strings to be stored in this file.

*Status* 0 indicates that duplicate keys are not allowed, 1 means that duplicate keys may occur.

On exit, the *OK* flag is set to *True* if the file was created successfully. If *OK* is false, there was no space on the disk for a new file.

### 4.2.7 CloseIndex

*CloseIndex* closes a TURBO Access *Index file*.

---

**ACCESS.BOX**:

```
procedure CloseIndex(var IndexF :  IndexFile);
```

**Parameter**:

*IndexF* The *Index file* variable to be closed. It must be of type *IndexFile*.

If you make any changes to an *Index file*, always call *CloseIndex* for that file before terminating your program. Failing to do so may cause data to be lost, not to mention possible corruption of the *Index file* structure.

### 4.2.8 AddKey

*AddKey* adds a key string to an *Index file.*

---

**ADDKEY.BOX:**

```
procedure AddKey(var IndexF   :   IndexFile;
                 var DataRef  :   Integer;
                 var Key                  );
```

**Parameters:**

*IndexF* The *Index file* to which the key is to be added.

*DataRef* The data record number to be associated with the key. Usually *DataRef* is a *Data file* record number returned by *AddRec.*

*Key* The key string. Since *Key* is an untyped parameter, you may pass string variables of any string type to *AddKey.* It is, however, up to you to ensure that the parameter is a string variable—other variables and string expressions are not allowed. If the *Key* parameter is longer than the maximum key length for *IndexF,* it will be truncated to the maximum length.

On exit, the *OK* flag is set to *True* if the key string was added successfully. *OK* returns *False* if you try to add a duplicate key when such keys are not allowed (that is, when the *Status* parameter in the call to *OpenIndex* or *MakeIndex* was 1.

### 4.2.9 DeleteKey

*DeleteKey* removes a key from an *Index file.*

---

**DELKEY.BOX:**

```
procedure DelKey(var IndexF   :   IndexFile;
                 var DataRef  :   Integer;
                 var Key                  );
```

**Parameters:**

*IndexF* *The Index file* from which the key is to be removed.

*DataRef* The data record number associated with the key to be deleted. If duplicate keys are not allowed in the *Index file, DataRef* need not be initialized. If duplicate keys are allowed, however, *DeleteKey* needs the data record to distinguish the keys from each other. To determine the data record number, you may, for example, use *SearchKey* in connection with *NextKey* and *PrevKey. DataRef* always returns the data record number of the key that was deleted.

*Key* The key to be deleted. The *Key* parameter must be a string variable. If it is longer than the maximum key length for *IndexF*, it will be truncated to the maximum length.

On exit, *OK* is set to *True* if the key was removed successfully. *OK* returns *False* if the key was not found. If duplicate keys are allowed, *OK* will return *False* if a matching data record number was not found, even though the key string existed.

### *4.2.10 FindKey*

*FindKey* returns the data record number associated with a key.

---

**GETKEY.BOX**:

```
procedure FindKey(var IndexF   :   IndexFile;
                  var DataRef  :   Integer;
                  var Key                 );
```

**Parameters**:

*IndexF* The *Index file* in which a search is to be conducted for the key.

*DataRef* If the key is found, its associated data reference is returned in this parameter.

*Key* The key string for which to search. The *Key* parameter must be a string variable. if it is longer than the maximum key length for *IndexF*, it will be truncated to the maximum length.

*FindKey* locates the entry in the index file that exactly matches the string passed as the *Key* parameter. If the *Index file* contains duplicate keys, *FindKey* will always locate the first key.

On exit, *OK* is set to *True* if a matching key is was found. Otherwise, *OK* is set to *False.*

### 4.2.11 SearchKey

*SearchKey* returns the data record number associated with the first entry in an *Index file* that is equal to or greater than a specific key value.

---

**GETKEY.BOX**:

```
procedure SearchKey(var IndexF :    IndexFile;
                    var DataRef :   Integer;
                    var Key         IndexFile;);
```

**Parameters**:

*IndexF* The *Index file* in which to search.

*DataRef* If the key is found, its associated data reference is returned in this parameter.

*Key* The key string for which to search. The *Key* parameter must be a string variable. If it is longer than the maximum key length for *IndexF*, it will be truncated to the maximum length.

*SearchKey* can be used to locate an entry in an *Index file* when only the first part of the key value is known. If the *Index file* contains duplicate keys, *SearchKey* will always locate the first key.

*OK* is always set to *True* on exit, unless no keys are greater than or equal to the search key. In that case, *OK* is set to *False*.

### 4.2.12 NextKey

*NextKey* returns the data reference associated with the next key in an *Index file.NextKey* also returns the key value in the *Key* parameter.

---

**GETKEY.BOX**:

```
procedure Next(var IndexF      :    IndexFile;
               var DataRef :        Integer;
               var Key                      );
```

**Parameters**:

*IndexF* An *Index file* that has been prepared for sequential processing by a call to *FindKey, SearchKey,* or *ClearKey.*

*DataRef* Returns the data reference associated with the key.

*Key* Returns the key read from the next index entry.

On exit, *OK* is set to *True* unless no next index entry exists. In that case, *OK* is set to *False.* When *OK* returns *False* (that is, when the pointer is at the end of the index), *NextKey* will return the first entry in the index if it is called again.

Before the very first call to *NextKey* for a given *Index file* or after the *Index file* is updated with *AddKey* or *DeleteKey,* one of the other index search functions (except *PrevKey*) must be called. The search functions establish the internal pointer used by *NextKey* and *PrevKey* for sequential processing.

### *4.2.13 PrevKey*

*PrevKey* returns the data reference associated with the preceding entry in an *Index file. PrevKey* also returns the key value in the *Key* parameter.

---

**GETKEY.BOX**:

```
procedure PrevKey(var IndexF   :    IndexFile;
                  var DataRef  :    Integer;
                  var Key                  );
```

**Parameters**:

*IndexF* An *Index file* that has been prepared for sequential processing by a call to *FindKey, SearchKey,* or *ClearKey.*

*DataRef* Returns the data reference associated with the key.

*Key* Returns the key from the preceding index entry.

On exit, *OK* is set to *True* unless no preceding index entry exists. In that case, *OK* is set to *False.* When *OK* returns *False* (that is, when the pointer is at the beginning of the index), *PrevKey* will return the last entry in the index if it is called again.

Before the very first call to *NextKey* for a given *Index file* or after the *Index file* is updated with *AddKey* or *DeleteKey,* one of the other index search functions (except *NextKey*) must be called. The search functions establish the internal pointer used by *NextKey* and *PrevKey* for sequential processing.

### 4.2.14 ClearKey

ClearKey sets the *Index file* pointer to the beginning or end of *IndexF*.

---

**ACCESS.BOX**:

```
procedure ClearKey(var IndexF    :    IndexFile);
```

**Parameter**:

*IndexF* An *Index file* that is prepared for sequential processing starting at the beginning or end.

Following a call to *ClearKey*, a call to *NextKey* will return the first entry in the *Index file*, and a call to *PrevKey* will return the last entry in the *Index file*.

When a TURBO Access *Index file* is processed sequentially it may be compared to a ring. When the *Index file* pointer is at the end of the file, a request to read the next entry will return the first entry in the file. Likewise, when the *Index file* pointer is at the beginning of the file, a request to read the preceding entry will return the last entry in the file. In fact, the beginning and the end are at the same point in the file.

# 5. PROGRAMMING EXAMPLES

The examples shown in this section demonstrate some common operations on a TURBO Access data base. In this case, the data base is very simple. The records in the *Data file* contain only two fields, and there is only one *Index file*. However, the principles are the same for more complex data bases with larger records and several *Index* and *Data files*.

## 5.1 Declaration Example

The following record type is used for records in the *Data file*:

```
type
  PhoneRec = record
               Status  :  Integer;
               Phone   :  String[15];
               Name    :  String[30];
             end;
```

The *Phone* field will be used as the key in the index. Note the inclusion of a *Status* field in the beginning of the record. If this field is set to zero when a record is added to the *Data file*, it will enable you to distinguish used records from deleted records when processing the *Data file* without reference to the *Index file* (for instance to rebuild a corrupted index).

The TURBO Access constants are defined as follows:

```
const
  MaxDataRecSize  =  49;   (* Max record size *)
  MaxKeyLen       =  15;   (* Max key size *)
  PageSize        =  16;   (* Page size *)
  Order           =   8;   (* B-tree Order*)
  PageStackSize   =   5;   (* Page buffer size *)
  MaxHeight       =   5;   (* Max B-tree height *)
```

*MaxDataRecSize* is the size of the *PhoneRec* type, and *MaxKeyLen* is the maximum length of the string contained in the *Phone* field.

The variables used in the examples are defined as follows:

```
var
  DatF      :  DataFile;
  IdxF      :  IndexFile;
  PRec      :  PhoneRec;
  PhNum     :  String[15];
  Ch        :  Char;
  DatRef,
  BufLen    :  Integer;
```

## 5.2 File Creation Example

The example shown below will create a *Data file* called **PHONE.DAT** and an *Index file* called **PHONE.IDX**. Note that the *SizeOf* function may be used to specify the size of a data record.

**Example:**

```
MakeFile(DatF,'PHONE.DAT',SizeOf(PRec));
if OK then MakeIndex(IdxF,'PHONE.IDX',15,0);
if OK then
begin
  CloseFile(DatF);
  CloseIndex(IdxF);
end
else
  Writeln('Cannot create data files');
```

## 5.3 Adding Records Example

The following example is used to add records to the data base. Since duplicate phone numbers are not allowed, the *Status* parameter in the call to *OpenIndex* is 0. A record is only added if a call to *FindKey* returns *False* in *OK*, indicating that the key is not in the index. If *AddKey* were to be called right away, the program would have to call *DeleteRec* in case of a duplicate key. Otherwise, the *Data file* would contain a record which was not in the index. Note that the *Status* field is set to zero before a new record is added so that active records can be differentiated from deleted records when the *Data file* is processed without reference to the *Index file.*

**Example:**

```
InitIndex;
OpenFile(DatF,'PHONE.DAT',SizeOf(PRec));
if OK then OpenIndex(IdxF,'PHONE.IDX',15,0);
if OK then
begin
   repeat
      Write('Phone number...: ');
      Readln(PhNum);
      PRec.Phone := PhNum;
      FindKey(IdxF,DatRef,PhNum);
      if OK then
      begin
         Writeln;
         Writeln('Already assigned');
      end
      else
      begin
         Write('Name.....: '); Readln(PRec.Name);
         PRec.Status := 0;
         AddRec(DatF,DatRef,PRec); AddKey(IdxF,DatRef,PRec.Phone);
      end;
      Writeln;
      Write('Add another (Y/N)? '); Readln(Ch);
      Writeln;
   until not (Ch in ['Y','y']);
   CloseFile(DatF);
   CloseIndex(IdxF);
end
else
   Writeln('Cannot open data files');
```

## 5.4 Find Records Example

This example shows how to look up records via the index. *FindKey* is used to search the index, and if the key exists *FindKey* returns the data reference. The data reference is then used in a call to *GetRec* to read the actual data record.

**Example:**

```
InitIndex;
OpenFile(DatF,'PHONE.DAT',SizeOf(PRec));
if OK then OpenIndex(IdxF,'PHONE.IDX',15,0);
if OK then
begin
   repeat
      Write('Phone number....: ');
      Readln(PhNum);
      FindKey(IdxF,DatRef,PhNum);
      if OK then
      begin
         GetRec(DatF,DatRef,PRec);
         Writeln('Name.....: ',PRec.Name);
      end
      else
      begin
         Writeln;
         Writeln('Key not found');
      end;
      Writeln;
      Write('Scan again (Y/N)? ');
      Readln(Ch);
      Writeln;
   until not (Ch in ['Y','y']);
   CloseFile(DatF);
   CloseIndex(IdxF);
end
else
   Writeln('Cannot open data files');
```

## 5.5 Deleting Records Example

The following example is used to delete records. *DeleteKey* is used to delete the specified key string. If the key is deleted successfully, *DeleteKey* returns the data record number associated with the key. This number is then used in a call to *DeleteRec* to delete the actual data record.

**Example:**

```
InitIndex;
OpenFile(DatF,'PHONE.DAT',SizeOf(PRec));
if OK then
   OpenIndex(IdxF,'PHONE.IDX',15,0);
if OK then
begin
   repeat
      Write('Phone number...: ');
      BufLen := 15;
      Readln(PhNum);
      DeleteKey(IdxF,DatRef,PhNum);
      if OK then
         DeleteRec(DatF,DatRef)
      else
      begin
         Writeln;
         Writeln('Key not found');
      end;
      Writeln;
      Write('Scan again (Y/N)? ');
      Readln(Ch);
      Writeln;
   until not (Ch in ['Y','y']);
   CloseFile(DatF);
   CloseIndex(IdxF);
end
else
   Writeln('Cannot open data files');
```

## 5.6 Sequential Seek Example

The *ClearKey* and *NextKey* routines may be used to process the data base sequentially as shown below. Note that *NextKey* returns *False* in *OK* when all keys in the index are processed.

**Example:**

```
InitIndex;
OpenFile(DatF,'PHONE.DAT',SizeOf(PRec));
if OK then OpenIndex(IdxF,'PHONE.IDX',15,0);
if OK then
begin
   ClearKey(IdxF);
   repeat
      NextKey(IdxF,DatRef,PhNum);
      if OK then
      begin
         GetRec(DatF,DatRef,PRec);
         Writeln(PRec.Phone,' :
                  16 - Length(PRec.Phone),PRec.Name);
      end;
   until not OK;
   CloseFile(DatF); CloseIndex(IDxF);
end
else
Writeln('Cannot close data files');
```

## 5.7 Status Field Example

A *Data file* can also be processed without reference to an *Index file.* However this requires that a status field (an integer) be included as the first field of each data record. When a record is added to the *Data file* (using the *AddRec* routine) the application program must manually set the status field to zero. When the *Data file* is processed sequentially, the status field will still be zero for a particular record if that record is not deleted. Other (non-zero) values indicate that the record is deleted. Note that processing starts at record number 1, which is actually the second record in the *Data file.* The first record, (that is, record number 0) is reserved by TURBO Access for various status information.

**Example:**

```
OpenFile(DatF,'PHONE.DAT',SizeOf(PRec));
if OK then
begin
   DatRef := 1;
   while DatRef < FileLen(DatF) do
   begin
         GetRec(DatF,DatRef,PRec);
         DatRef:=DatRef+1;
      if PRec.Status = 0 then
         Writeln(PRec.Phone," : 16 - Length(PRec.Phone),
                  PRec.Name                              );
   end;
CloseFile(DatF);
end
else
Writeln('Cannot open data file');
```

## 5.8 Restoring Index Example

The following example shows how to rebuild a corrupted *Index file*. The program reads the *Data file* sequentially and adds the phone number of each active record to a new *Index file*. If a duplicate key is encountered, an error message is output and the data record is deleted.

**Example:**

```
InitIndex;
OpenFile(DatF,'PHONE.DAT',SizeOf(PRec));
if OK then
begin
   MakeFile(IdxF,'PHONE.IDX',15,0);
   if OK then
   begin
      DatRef := 1;
      while DatRef<FileLen(DatF) do
      begin
         GetRec(DatF,DatRef,PRec);
         if PRec.Status = 0 then
         begin
            AddKey(IdxF,DatRef,PRec.Phone);
            if not OK then
            begin
               Writeln('Duplicate key. Record ',DatRef);
               DeleteRec(DatF,DatRef);
            end;
         end;
         DatRef:=DatRef+1;
      end;
   end
   else
      Writeln('Cannot create index file');
end
else
   Writeln('Cannot open data file');
```

# 6. ERROR HANDLING

TURBO Access routines generate two types of errors: non-fatal errors and fatal errors. Fatal errors cause the program to terminate, whereas non-fatal errors are reported to the program.

Non-fatal error conditions are reported through a boolean variable called *OK*, which is automatically declared by the **DTAMAN** module. For instance *OpenFile* returns *False* in *OK* if the specified file was not found and *FindKey* returns *False* if the key string was not found.

If a fatal error occurs, a routine called *TalOcheck* is invoked. *TalOcheck* is located in the ACCESS file. It prints an error code, a file name, and a record number, and then terminates the program. The following is an example of an error printout:

**Example:**

```
TURBO-file I/O error 10
   File A:CUST.DAT Record 103
   Program terminated
```

A TURBO Access fatal error is actually equivalent to a TURBO Pascal I/O error. Possible error codes are therefore the same as those listed in the *TURBO Pascal Reference Manual* (note that TURBO Access outputs the error code in decimal, whereas TURBO Pascal outputs it in hexadecimal).

In general, fatal errors occur only from corrupted *Data* and/or *Index file*. However, a fatal error will also occur if you try to expand a *Data file* or an *Index file* when there is insufficient space on the disk.

NOTES

# 7. SAMPLE PROGRAM

This chapter provides a sample program which illustrates how to use TURBO Access in a typical situation.

## 7.1 Description Of Sample Program

The file BTREE.PAS demonstrates how to use TURBO Pascal file to create and maintain a simple customer data base. BTREE allows you to add, find, view, edit, delete, and list customers of a predefined type (see the *CustRec* type definition below).

BTREE maintains three files: a *Data file* (CUST.DAT), a customer code index (CUST.IXC) and a name index (CUST.IXN). The customer code index does not allow duplicate keys, but the name index does. When the database program is run for the first time, it will automatically create an empty data base.

The main menu offers three functions: **U**pdate, **L**ist, and **Q**uit. **U**pdate is used to add, find, view, edit, and delete customers. **L**ist is used to List customers, and **Q**uit is used to terminate the program.

On the **U**pdate menu, the **A**dd function is used to add new customers. **F**ind is used to locate a customer, either by customer code or by last (and first) name. To Search for a specific customer code, simply enter the desired code when the cursor is located in the customer code field. If the code is found, the customer data is displayed. At this point, you may, if you wish, **E**dit or **D**elete it. To search for a name, enter an empty customer code. Then enter the last name and, if desired, the first name. (Note that if a first name is specified, the first 15 characters of the last name must be entered in full.) The scan will locate the first customer with the specified name, or if no exact match is found, the first customer that follows the specified name. You may then use **N**ext and **P**revious to move forward and backward in alphabetical order. Once you have located the desired customer, enter **Q**uit. You may then **E**dit or **D**elete the record shown on the screen, or just view it and leave it unchanged.

**L**ist is used to List customers. The listings will show the customer code, the name, and the company. The Listing may be output to the **P**rinter or to the **S**creen, and may be either **U**nsorted or sorted by customer **C**ode or **N**ame.

For further information, study this sample source code.

On systems with only 64K of RAM you will find that the file BTREE.PAS is too large to allow the compiler sufficient work space. This results in a compiler overflow error condition. The problem is solved by splitting BTREE.PAS into a main file and an insert file.

## 7.2 BTREE.PAS / BTREE.INC Sample Source

```
program DataBase;
(*$A+,C-,R-,V-*)

(*****************************************************************************)
(*                                                                           *)
(*                          TURBO-access version 1.00                        *)
(*                                                                           *)
(*                             DATABASE example                              *)
(*                                                                           *)
(*                             Copyright© 1984 by                            *)
(*                               BORLAND Int.                                *)
(*                                                                           *)
(*                                                                           *)
(*                                                                           *)
(*                                                                           *)
(*****************************************************************************)


label Stop;
const
(* data record Size definition *)
   CustRegSize      = 342;                 (*customer record Size *)
(* TURBO-file constants *)
   MaxDataRecSize   = CustRegSize;         (* max record Size *)
   MaxKeyLen        = 25;                  (* max key Size *)
   PageSize         = 16;                  (* page Size *)
   Order            =  8;                  (* half page Size *)
   PageStackSize    =  5;                  (* page buffer Size *)
   MaxHeight        =  5;                  (* max B-tree height *)
var
   NoOfRecs         : Integer;
(* include TURBO-file modules *)
(*$I ACCESS.BOX*)
(*$I GETKEY.BOX*)
(*$I ADDKEY.BOX*)
(*$I DELKEY.BOX*)

type
   Str5     =  string[5];
   Str10    =  string[10];
   Str15    =  string[15];
   Str25    =  string[25];
   Str80    =  string[80];
   AnyStr   =  string[255];
```

```
(* character set type *)
   CharSet = set of Char;
(* customer record definition *)
CustRec = record
                CustStatus    :   Integer;          (* CustStatus *)
                CustCode      :   string[15];       (* customer code *)
                EntryDate     :   string[8];        (* entry date *)
                FirstName     :   string[15];       (* first name *)
                LastName      :   string[30];       (* last name *)
                Company       :   string[40];       (* company *)
                Addr1         :   string[40];       (* Address 1 *)
                Addr2         :   string[40];       (* Address 2 *)
                Phone         :   string[15];       (* Phone number *)
                PhoneExt      :   string[5];        (* extension *)
                Remarks1      :   string[40];       (* remarks 1 *)
                Remarks2      :   string[40];       (* remarks 2 *)
                Remarks3      :   string[40];       (* remarks 3 *)
             end;
var
(* global variables *)
   DatF                       :   DataFile;
   CodeIndexFile,
   NameIndexFile              :   IndexFile;
   Ch                         :   Char;
function UpcaseStr(S : Str80) : Str80;
var
   P : Integer;
begin
   for P := 1 to Length(S) do
      S[P] := Upcase(S[P]);
   UpcaseStr := S;
end;
(* ConstStr returns a string with N characters of value C *)
function ConstStr(C : Char; N : Integer) : Str80;
var
   S : string[80];
begin
   if N < 0 then
      N := 0;
   S[0] := Chr(N);
   FillChar(S[1],N,C);
   ConstStr := S;
end;
```

```
(* Beep sounds the terminal bell or beeper *)
procedure Beep;
begin
   Write(^G);
   end;
procedure InputStr(var    S       : AnyStr;
                          L,X,Y   : Integer;
                          Term    : CharSet;
                   var    TC      : Char);
const
   UnderScore='_';
var
   P : Integer;
   Ch : Char;
begin
   GotoXY(X + 1,Y + 1); Write(S,ConstStr(UnderScore,L - Length(S)));
   P := 0;
   repeat
      GotoXY(X + P + 1,Y + 1); Read(Kbd,Ch);
      case Ch of
#32..#126 : if P < L then
                                 begin
                                    Length(S) = L then
                                       Delete(S,L,1);
                                    P := P + 1;
                                    Insert(Ch,S,P);
                                    Write(Copy(S,P,L));
                                 end
                                 else Beep;
            ^S          :        if P > 0 then
                                    P := P - 1
                                 else Beep;
            ^D          :        if P < Length(S) then
                                    P := P + 1
                                 else Beep;
            ^A          :        P := 0;
            ^F          :        P := Length(S);
            ^G          :        if P < Length(S) then
                                 begin
                                    Delete(S,P + 1,1);
                                    Write(Copy(S,P + 1,L),UnderScore);
                                 end;
            ^H,#127     :        if P > 0 then
```

```
                                begin
                                   Delete(S,P,1);
                                   Write(↑H,Copy(S,P,L),UnderScore);
                                   P := P - 1;
                                end
                                else Beep;
                  ↑Y        :   begin
                                   Write(ConstStr(UnderScore,Length(S) P));
                                   Delete(S,P + 1,L);
                                end;
        else
           if not (Ch in Term) then Beep;
        end;{of case}
    until Ch in Term;
    P := Length(S);
    GotoXY(X + P + 1,Y + 1);
    Write(" :L-P);
    TC := Ch;
end;

procedure Select(     Prompt    :   Str80;
                      Term      :   CharSet;
                  var TC        :   Char    );
var
    Ch : Char;
begin
    GotoXY(1,23); Write(Prompt,'? ' ); ClrEol;
    repeat
       Read(Kbd,Ch);
       TC := Upcase(Ch);
       if not (TC in Term) then
          Beep;
    until TC in Term;
    Write(Ch);
end;
(*ClearFrame clears the display frame, I.E. Lines 3 to 20 *)
procedure ClearFrame;
var
    I : Integer;
begin
    for I := 3 to 20 do
    begin
       GotoXY(1,I + 1); ClrEol ;
    end;
end;
(* OutForm displays the entry form on the screen *)
```

```
procedure OutForm;
begin
   GotoXY(7,5); Write('Code :');
   GotoXY(29,5); Write('Date :');
   GotoXY(1,7); Write('First name :');
   GotoXY(29,7); Write('Last name :');
   GotoXY(4,9); Write('Company :');
   GotoXY(2,10); Write('Address 1 :');
   GotoXY(2,11); Write('Address 2 :');
   GotoXY(6,13); Write('Phone :');
   GotoXY(29,13); Write('Extension :');
   GotoXY(2,15); Write('Remarks 1 :');
   GotoXY(2,16); Write('Remarks 2 :');
   GotoXY(2,17); Write('Remarks 3 :');
end;
(* ClearForm clears all fields in the entry form *)
procedure ClearForm;
begin
   GotoXY(13,5); Write(" :15);
   GotoXY(35,5); ClrEol;
   GotoXY(13,7); Write(" :15);
   GotoXY(40,7); ClrEol;
   GotoXY(13,9); ClrEol;
   GotoXY(13,10); ClrEol;
   GotoXY(13,11); ClrEol;
   GotoXY(13,13); Write(" :15);
   GotoXY(40,13); ClrEol;
   GotoXY(13,15); ClrEol;
   GotoXY(13,16); ClrEol;
   GotoXY(13,17); ClrEol;
end;
procedure InputCust(var Cust : CustRec);
const
   Term : CharSet = [^E,^I,^M,^X,^Z];
var
   L : Integer;
   TC : Char;
begin
   L := 1;
   with Cust do
   repeat
      case L of
         1 : InputStr(CustCode,15,12,4,Term,TC);
         2 : InputStr(EntryDate,8,34,4,Term,TC);
         3 : InputStr(FirstName,15,12,6,Term,TC);
         4 : InputStr(LastName,30,39,6,Term,TC);
         5 : InputStr(Company,40,12,8,Term,TC);
```

```
            6 : InputStr(Addr1,30,12,9,Term,TC);
            7 : InputStr(Addr2,30,12,10,Term,TC);
            8 : InputStr(Phone,15,12,12,Term,TC);
            9 : InputStr(PhoneExt,5,39,12,Term,TC);
            10 : InputStr(Remarks1,40,12,14,Term,TC);
            11 : InputStr(Remarks2,40,12,15,Term,TC);
            12 : InputStr(Remarks3,40,12,16,Term,TC);
         end;
         if (TC = ↑I) or (TC = ↑M) or (TC = ↑X) then
            if L = 12 then
               L := 1
            else L := L + 1
         else
            if TC = ↑E then
               if L = 1 then
                  L := 12
               else L := L - 1;
   until (TC = ↑M) and (L = 1) or (TC = ↑Z);
end;
(* OutCust displays the customer data contained in Cust *)
procedure OutCust(var Cust : CustRec);
begin
with Cust do
begin
   GotoXY(13,5); Write(CustCode,' :15 - Length(CustCode));
   GotoXY(35,5); Write(EntryDate); ClrEol ;
   GotoXY(13,7); Write(FirstName,' :15 - Length(FirstName));
   GotoXY(40,7); Write(LastName); ClrEol;
   GotoXY(13,9); Write(Company); ClrEol;
   GotoXY(13,10); Write(Addr1); ClrEol;
   GotoXY(13,11); Write(Addr2); ClrEol;
   GotoXY(13,13); Write(Phone,' :15 - Length(Phone));
   GotoXY(40,13); Write(PhoneExt); ClrEol;
   GotoXY(13,15); Write(Remarks1); ClrEol;
   GotoXY(13,16); Write(Remarks2); ClrEol;
   GotoXY(13,17); Write(Remarks3); ClrEol;
   end;
end;
function KeyFromName(LastNm : Str15; FirstNm : Str10) : Str25;
const
   Blanks='                    ';
begin
   KeyFromName := UpcaseStr(LastNm)+
                  Copy(Blanks,1,15 - Length(LastNm))+
                  UpcaseStr(FirstNm);
end;
```

```
(* Update is used to update the data base *)
procedure Update;
var
   Ch : Char;
(* Add is used to add customers *)
procedure Add;
var
   DataF          :   Integer;
   Ccode          :   string[15];
   KeyN           :   string[25];
   Cust           :   CustRec;
begin
   with Cust do
   begin
      FillChar(Cust,SizeOf(Cust),0);
      repeat
         InputCust(Cust);
         Ccode := CustCode;
         FindKey(CodeIndexFile, DataF,Ccode);
         if OK then
         begin
            GotoXY(6,19);
            Write('ERROR : Duplicate customer code');
            Beep;
         end;
      until not OK;
      AddRec(DatF,DataF,Cust);
      AddKey(CodeIndexFile, DataF,CustCode);
      KeyN := KeyFromName(LastName,FirstName);
      AddKey(NameIndexFile, DataF,KeyN);
      GotoXY(6,19); ClrEol;
   end;
end;
(* Find is used to find, edit and delete customers *)
procedure Find;
var
   D,L,I          :   Integer;
   Ch,
   TC             :   Char;
   Ccode,
   PCode,
   FirstNm        :   string[15];
   KeyN,
   PNm            :   string[25];
   LastNm         :   string[30];
   Cust           :   CustRec;
```

```
begin
  if UsedRecs(DatF) > 0 then
  begin
    Ccode := '';
    repeat
      InputStr(Ccode,15,12,4,[↑M,↑Z],TC);
      if Ccode <> '' then
      begin
        FindKey(CodeIndexFile,D,Ccode);
        if OK then
        begin
          GetRec(DatF,D,Cust);
          OutCust(Cust);
        end
        else
        begin
          GotoXY(6,19);
          Write('ERROR : Customer code not found'); Beep;
        end;
      end;
    until OK or (Ccode = '');
    GotoXY(6,19); ClrEol;
    if Ccode = '' then
    begin
      L := 1;
      FirstNm := '';
      LastNm := '';
      repeat
        case L of
          1 : InputStr(FirstNm,15,12,6,[↑I,↑M,↑Z],TC);
          2 : InputStr(LastNm,30,39,6,[↑I,↑M,↑Z],TC);
        end;
        if (TC = ↑I) or (TC = ↑M) then
          L := 3 - L;
        until (TC = ↑M) and (L = 1) or (TC = ↑Z);
        KeyN := KeyFromName(LastNm,FirstNm);
        SearchKey(NameIndexFile, D,KeyN);
        if not OK then
          PrevKey(NameIndexFile,D,KeyN);
        repeat
          GetRec(DatF,D,Cust);
          OutCust(Cust);
          Select('Find : N)ext, P)revious, Q)uit',['N','P','Q'],Ch);
          case Ch of
            'N' : repeat NextKey(NameIndexFile, D,KeyN) until OK;
            'P' : repeat PrevKey(NameIndexFile, D,KeyN) until OK;
          end;
```

```
            until Ch = 'Q';
        end;
        Select('Find : E)dit, D)elete, Q)uit',['E','D','Q'],Ch);
        with Cust do
        case Ch of
            'E' : begin
            PCode := CustCode;
                    PNm := KeyFromName(LastName,FirstName);
                    repeat
                        InputCust(Cust);
                        if CustCode = PCode then
                            OK := false
                        else
                        begin
                            Ccode := CustCode;
                            FindKey(CodeIndexFile, I,Ccode);
                            if OK then Beep;
                        end;
                    until not OK;
                    PutRec(DatF,D,Cust);
                    if CustCode <> PCode then
                    begin
                        DeleteKey(CodeIndexFile, D,PCode);
                        AddKey(CodeIndexFile, D,CustCode);
                    end;
                    KeyN := KeyFromName(LastName,FirstName);
                    if KeyN <> PNm then
                    begin
                        DeleteKey(NameIndexFile, D,PNm);
                        AddKey(NameIndexFile, D,KeyN);
                    end;
                end;
            'D' : begin
                    DeleteKey(CodeIndexFile,D,CustCode);
                    KeyN := KeyFromName(LastName,FirstName);
                    DeleteKey(NameIndexFile,D,KeyN);
                    DeleteRec(DatF,D);
            end;
        end;
    end { of UsedRecs(DatF) > 0 .. }
    else Beep;
end;
```

```
begin(* Update*)
   OutForm;
   repeat
      Select('Update : A)dd, F)ind, Q)uit',['A','F','Q'],Ch);
      case Ch of
         'A' : Add;
         'F' : Find;
      end;
      if Ch <> 'Q' then
      begin
         GotoXY(60,2); Write(UsedRecs(DatF) :5);
         ClearForm;
      end;
   until Ch = 'Q';
end;
(* List is used to list customers *)
procedure List;
label Escape;
var
   D,L,LD      :   Integer;
   Ch,CO,CS    :   Char;
   Ccode       :   string[15];
   KeyN        :   string[25];
   Name        :   string[35];
   Cust        :   CustRec;
begin
   Select('Output device : P)rinter, S)creen',['P','S'],CO);
   Select('Sort by : C)ode, N)ame, U)nsorted',['C','N','U'],CS);
   GotoXY(1,23); Write('Press <Esc> to abort'); ClrEol;
   ClearKey(CodeIndexFile);
   ClearKey(NameIndexFile);
   D := 0;
   LD:=FileLen(DatF)-1;
   L:=3;
   repeat
      if KeyPressed then
      begin
         Read(Kbd,Ch);
         if Ch = #27 then
            goto Escape;
         end;
```

```
case CS of
   'C' : NextKey(CodeIndexFile,D,Ccode);
   'N' : NextKey(NameIndexFile,D,KeyN);
   'U' : begin
            OK := false;
            while (D < LD) and not OK do
            begin
               D := D + 1;
               GetRec(DatF,D,Cust);
               OK := Cust.CustStatus = 0;
            end;
         end;
end;
if OK then
   with Cust do
   begin
      if CS <> 'U' then
         GetRec(DatF,D,Cust);
      Name := LastName;
      if FirstName <> '' then
         Name := Name + ', ' + FirstName;
      if CO = 'P' then
      begin
         Write(Lst,CustCode,' :16 - Length(CustCode));
         Write(Lst,Name,' :36 - Length(Name));
         Writeln(Lst,Copy(Company,1,25));
      end
      else
      begin
         if L = 21 then
         begin
            GotoXY(1,23);
            Write('Press < RETURN> to continue');
            Write(' or <Esc> to abort');
            ClrEol;
            repeat
               Read(Kbd,Ch)
            until (Ch = ^M) or (Ch = #27);
            if Ch = #27 then
               goto Escape;
            GotoXY(1,23);
            Write('Press <Esc> to abort'); ClrEol;
            ClearFrame;
            L := 3;
         end;
         GotoXY(1,L + 1); Write(CustCode);
```

```
                GotoXY(17,L + 1); Write(Name);
                GotoXY(53,L + 1); Write(Copy(Company,1,25));
                L := L + 1;
              end; { of with Cust do .. }
        end; { of if OK .. }
    until not OK;
    if CO = 'S' then
    begin
        GotoXY(1,23); Write('Press <RETURN>'); ClrEol;
        repeat
            Read(Kbd,Ch)
        until Ch = ↑M;
    end;
    Escape :
end;
(* Main program *)
begin
    ClrScr ;
    Writeln(ConstStr('-',79));
    Writeln('TURBO-file Customer Database');
    Writeln(ConstStr('-',79));
    GotoXY(1,22); Writeln(ConstStr('-',79));
    Writeln;
    Write(ConstStr('-',79)); GotoXY(1,4);
    InitIndex;
    OpenFile(DatF,'CUST.DAT',CustRegSize);
    if OK then
        OpenIndex(CodeIndexFile,'CUST.IXC',15,0);
    if OK then
        OpenIndex(NameIndexFile,'CUST.IXN',25,1);
    if not OK then
    begin
        Select('Data files missing. Create new files (Y/N)', ['Y','N'], Ch);
        if Ch = 'Y' then
        begin
            MakeFile(DatF,'CUST.DAT',CustRegSize);
            MakeIndex(CodeIndexFile,'CUST.IXC',15,0);
            MakeIndex(NameIndexFile,'CUST.IXN',25,1);
        end
        else goto Stop;
    end;
```

```
    GotoXY(60,2); Write(UsedRecs(DatF):5,' Records in use');
    repeat
      Select('Select : U)pdate, L)ist, Q)uit', ['U','L','Q'], Ch);
      case Ch of
        'U' : Update;
        'L' : List;
      end;
      if Ch <> 'Q' then ClearFrame;
    until UpCase(Ch) = 'Q';
    CloseFile(DatF);
    CloseIndex(CodeIndexFile) ;
    CloseIndex(NameIndexFile) ;
    Stop :ClrScr;
end.
```

# PART II

# The TURBO Sort System

# 8. INTRODUCTION TO TURBO SORT

As a programmer, you can spend much time developing routines to sort your data. The TURBO Sort system lets you rid yourself of the details of the steps required to sort your data, and concentrate on other important aspects of your application program while providing your users with a versatile sorting utility.

This chapter provides a brief overview of the capabilities of TURBO Sort.

## 8.1 This is TURBO Sort

TURBO Sort may be thought of as a 'black box' which will take care of all your sorting needs with a minimum of coding on your part. TURBO Sort uses the Quicksort method to ensure fast and efficient sorting, and will sort any **type** of data. TURBO Sort's virtual memory management means that you are not limited to sorting in memory; if your data requires more memory than is available for sorting, the disk will automatically be used as an extension of memory.

TURBO Sort is supplied on the disk in readable source code which you are free to use any way you like. You can include this file in your own TURBO Pascal programs, as explained later in this manual, without ever taking a look at the source code, and without worrying about how it works. You may study the source code to learn from it, or you may even make your own changes to it. But remember that if you do that, you are as much on your own as if you had written the entire program yourself, and our support staff will have no way of helping you.

## 8.2 About the Text

This part of the manual teaches you how to use TURBO Sort in your own TURBO Pascal programs. It provides all the information you need to perform sorting, but no more. We are not trying to keep the inner workings of TURBO Sort a secret; on the contrary, TURBO Sort is available to you in source code, but discussing it here is unnecessary and would only cause confusion.

Chapter 9 shows you the few and simple things you must do to sort single data items.

Chapter 10 discusses some more advanced uses: sorting on multiple keys and sorting of different data items in the same program.

Because TURBO Sort is a tool you use in TURBO Pascal programs of your own creation, we have assumed that you are already familiar with TURBO Pascal. Therefore, the text makes no attempt to explain the Pascal language or the extensions found in TURBO Pascal.

## 8.3 Files on the Toolbox Diskette

The TURBO Toolbox distribution diskette contains a number of files corresponding to different TURBO tools. All tools have the extension .BOX. All sample programs are have the extension .PAS. You should make a copy of the distribution diskette before you go to work to ensure that you always have the original, untouched versions of your TURBO tools.

The following files belong to the TURBO Sort system:

| | |
|---|---|
| **SORT.BOX** | Contains the *TurboSort* function and related procedures and functions. |
| **SORT1.PAS** | Sample TURBO Pascal program using TURBO Sort. Sorts data from the CUSTOMER.DTA file. |
| **SORT2.PAS** | Sample TURBO Pascal program using TURBO Sort. Sorts data from the CUSTOMER.DTA file on a single key and from the STOCK.DTA files on multiple keys. |
| **CUSTOMER.DTA** | Data for SORT1 and SORT2 above. |
| **STOCK.DTA** | Data for SORT2 above. |

# 9. USING TURBO SORT

This chapter provides explanations and examples illustrating how to use TURBO Sort. Read this information carefully, for this is the basis for all TURBO Sort operations.

## 9.1 How TURBO Sort Works

TURBO Sort is a **function** of type *Integer*. The TURBO Sort function is called with one parameter, as follows:

```
SortResult := TurboSort(ItemSize);
```

where *SortResult* is an *Integer* variable and *ItemSize* is an *Integer* expression giving the size (in bytes) of the data items which are to be sorted.

TURBO Sort divides its work into three phases:

- the **input** phase
- the **sorting** phase
- the **output** phase

In the input phase, TURBO Sort calls the procedure *Inp*, which you must write. This procedure inputs data to be sorted and passes it to TURBO Sort, one item at a time. Input data may be read from a file, it may be data already in memory, or it may be produced by the *Inp* procedure itself. In short, since you write the *Inp* procedure yourself, you are free to obtain input by any means. Furthermore, input may be data of any **type**, because it is passed to TURBO Sort as an untyped parameter. *Inp* is called only once. When it finishes, the sorting phase starts.

Since TURBO Sort knows nothing of the type of data being sorted, it relies on the boolean function *Less* (which you must write) to determine which of two data items is the smaller. The *Less* function is called repeatedly during the sorting phase. When sorting is finished, TURBO Sort enters the output phase.

In the output phase, TURBO Sort calls the procedure *OutP* (which you also must write). *OutP* gets the sorted data one item at a time, allowing you to do with it whatever you please; for example, write it on a file, put it in memory for further processing, print it, and so forth. Like *Inp*, *OutP* gives you complete freedom in dealing with your data, because it is a procedure of your own design. *OutP* is called only once, and when it finishes, TURBO Sort terminates.

When the TURBO Sort function terminates, it produces an integer value which indicates whether sorting went well, or aborted with an error.

### 9.1.1 Data Item Size

The parameter passed to TURBO Sort is the size (in number of bytes) of the data item you want to sort. The TURBO Pascal standard function *SizeOf* will give you this information, as shown in the following example:

```
ItemSize := SizeOf(DataItem);
```

where *ItemSize* is an integer variable and *DataItem* is the variable you want to sort (or the **type** of that variable).

### 9.1.2 Use of Memory

TURBO Sort will automatically allocate space on the *Heap* for sorting. TURBO Sort allocates *MaxAvail* minus 2K bytes (to ensure ample space on the stack for local variables and parameters). If your *Inp*, *Less*, or *OutP* subprograms require *Heap* space, you must allocate space for them **before** calling *TurboSort* by using the standard procedures *New* or *GetMem*.

The *minimum* size required for sorting is:

3 * *ItemSize*

or

3 * 128 bytes

whichever is larger. If less space is available, TURBO Sort aborts and returns the error value 3.

TURBO Sort will perform sorting entirely within memory if space allows. Only if there is insufficient space for internal sorting will TURBO Sort's virtual memory management be activated. Then, the disk will work as an extension of memory.

### 9.1.3 Maximum Sort Size

The maximum number of records TURBO Sort will handle is 32767 (*MaxInt*). If more records are to be found, TURBO Sort aborts with error code 9.

## 9.2 A Sample Program Using TURBO Sort

Let us suppose that you have a file of customers that you want to sort. The following illustrates how you might write a sample program that reads data from such a file, sorts it by customer number, and outputs to the screen.

The file CUSTOMER.DTA on the distribution diskette contains 100 records of the **type** defined below, and is used by the example for input.

You start your program as usual with the **type** definition part and the variable declaration part:

```
program SortExampleOne;

type
   CustRec = record
                Number: integer;
                Name:string[30];
                Addr:string[20];
                City:string[12];
                State:string[3];
                Zip:string[5];
             end;

var
   CustFile: file of CustRec;
   Customer: CustRec;

{$ISORT.BOX}
```

*CustRec* is the data item which is to be read from the file, sorted, and finally output to the screen.

The {$ISORT.BOX} statement causes the compiler to include the file SORT.BOX during compilation. This file contains the *TurboSort* function and related declarations.

After making these declarations, you are ready to write the three simple subprograms *Inp*, *Less*, and *OutP*. The following subsections explain each of these.

### 9.2.1 The Inp Procedure

Because the *Inp* procedure is called from the *TurboSort* function in the SORT.BOX file, it must be **forward** declared prior to the declaration of *TurboSort*. The SORT.BOX file contains the necessary declaration. Your *Inp* procedure should look like this:

```
procedure Inp;
begin
   repeat
      Read(CustFile,Customer);
      SortRelease(Customer);
   until EOF(CustFIle);
end;
```

The *Inp* procedure is called only once from *TurboSort*. It then reads records from the data file *CustFile*, and passes them on for sorting with calls to the procedure *SortRelease* (which is also included in the SORT.BOX file). This process is repeated until end-of-file is reached. The parameter to *SortRelease* is untyped, which means that you can pass data of any type to *SortRelease*.

You have now read all of your input data and passed it on for sorting. Clearly, data could be obtained from other sources than files.

### 9.2.2 The Less Function

Like *Inp*, the *Less* function is **forward** declared in the SORT.BOX file. It is declared as a *Boolean* function with two untyped parameters *X* and *Y*. The type and the parameters must not be repeated in your declaration of *Less*, which should look like this:

```
function Less;
var
   FirstCust:CustRec absolute X;
   SecondCust: CustRec absolute Y;
begin
   Less := FirstCust.Number < SecondCust.Number;
end;
```

The *Less* function receives two memory addresses in the parameters *X* and *Y*. These are the addresses of the first byte of the first to data items *TurboSort* is to compare. You then declare two variables 'on top' of these data items by declaring the variables as **absolute** at addresses *X* and *Y*. The variables then contain the data items to be compared.

Comparing the variables on the desired criteria is now simple. In the example, we compare the customer number. The customer number thus becomes the sorting *key*. We could have just as simply sorted on the *Name*, the *Zip* code, or any other field in the record, or even could have sorted on multiple keys by comparing more fields. But let's keep the example simple.

*Less* is called repeatedly by *TurboSort*, once every time two data items are to be compared. When *TurboSort* has finished sorting, the output procedure *OutP* is called.

### 9.2.3 The OutP Procedure

Like *Inp*, the *OutP* procedure is called from the *TurboSort* function and must therefore be **forward** declared in the SORT.BOX file.

```
procedure OutP;
var
   I:Integer;
begin
   repeat
      SortReturn(Customer);
      with Customer do
      begin
         Write(Number,' ',Name,' ');
         for I := Length(Name) to 30 do Write(' ');
         Write(Addr);
         for I := Length(Addr) to 20 do Write(' ');
         Write(City);
         for I := Length(City) to 12 do Write(' ');
         WriteLn(State,' ',Zip);
      end;
   until SortEOS;
end;
```

The *OutP* procedure is called only once from *TurboSort*. It then calls the *SortReturn* procedure which is part of the SORT.BOX file. *SortReturn* returns one data item in its parameter (again untyped). This data item can now be output. The process is repeated until the *SortEOS* function (also part of the SORT.BOX file) returns *True*. The basic functionality of the output procedure reads as follows:

```
repeat
   SortReturn(Customer);
until SortEOS;
```

The **with** Customer **do**... statement writes the sorted customer records to the screen, one record at a time, with each field left justified. The output could instead be sent to a file, a printer, or anywhere else you can specify.

### 9.2.4 The Main Program

In the main program, you first prepare the input file for reading by using the *Assign* and *Reset* standard procedures. You then start the sorting with a call to the *TurboSort* function. In the example, this is done in a *WriteLn* statement which will print the value of the *TurboSort* function on the screen when sorting is over. This value tells you whether everything went well, or sorting was aborted with an error.

```
begin {program SortExampleOne}
  Assign(CustFile,'CUSTOMER.DTA');
  Reset(Custfile);
  WriteLn(TurboSort(SizeOf(CustRec)));
end.
```

The parameter to *TurboSort* is an *Integer* expression giving the size (in bytes) of the data item to be sorted. The standard function *SizeOf* is convenient to use because it returns the size of its argument, which may be the identifier of either the **type** or the variable to be sorted.

### 9.2.5 TURBO Sort Termination Information

The value of the function *TurboSort* (which is printed by the sample program), indicates certain error conditions, as follows:

**0** All went well.

**3** Not enough memory available for sorting. The minimum size is three times the size of the data item to be sorted, or 3 * 128 bytes, whichever is larger.

**8** Illegal item length. Item length must be >=2.

**9** More than *MaxInt* records input for sort.

**10** An error occured during sorting. This may mean a bad disk or that the disk became full.

**11** Read error during sort. Probably due to a bad disk.

**12** File creation error. The directory may be full, or you may try to access a non-existing directory (MS-DOS/PC-DOS v. 2).

## 9.3 Complete Example

This section lists the complete example described above. The file SORT1.PAS on the distribution diskette contains the example.

```
program SortExampleOne { Customer File };

type
   CustRec = record
         Number:      integer;
         Name:        string[30];
         Addr:        string[20];
         City:        string[12];
         State:       string[3];
         Zip:         string[5];
            end;
var
   CustFile: file of CustRec;
   Customer: CustRec;
{$ISORT.BOX}
procedure Inp; { forward declared in SORT.BOX }
begin
   repeat
      Read(CustFile,Customer);
      SortRelease(Customer);
   until EOF(CustFIle);
end;

function Less; { boolean function with two parameters, }
               { X and Y, forward declared in SORT.BOX }
var
   FirstCust:CustRec absolute X;
   SecondCust: CustRec absolute Y;
begin
   Less := FirstCust.Number < SecondCust.Number;
end;
procedure OutP; { forward declared in SORT.BOX }
   var
      I:Integer;
   begin
      repeat
         SortReturn(Customer);
         with Customer do
```

```
        begin
          Write(Number,' ',Name,' ');
          for I := Length(Name) to 30 do Write(' ');
          Write(Addr);
          for I := Length(Addr) to 20 do Write(' ');
          Write(City);
          for I := Length(City) to 12 do Write(' ');
          WriteLn(State,' ',Zip);
        end;
    until SortEOS;
end;
begin
    Assign(CustFile,'CUSTOMER.DTA');
    Reset(Custfile);
    WriteLn(TurboSort(SizeOf(CustRec)));
end.
```

# 10. ADVANCED SORTING

The example presented in the previous chapter deals with the simple case of sorting one type of data, and sorting on one key. It was a useful example, for you will often want to sort in this manner. However, you may also want to be able to sort different kinds of data, or sort on multiple keys. Such advanced sorting is discussed in this chapter.

## 10.1 Sorting Different Data

We will use the example from the previous chapter as a basis for a new program which can sort both the customer data we already know and items in a stock list. The first thing to do is to add the definition of a new **type** to the program declaration, as follows:

```
program SortExampleTwo (Customer File and Stock File);

type
  CustRec = record
      Number:     integer;
      Name:       string[30];
      Addr:       string[20];
      City:       string[12];
      State:      string[3];
      Zip:        string[5];
    end;

  ItemRec = record
      Number:     integer;
      Descrip:    string[30];
      InStock:    integer;
      Price:      real;
    end;
```

The new **type** *ItemRec* defines a data record which will hold information about items in the stock list. The file STOCK.DTA contains 100 records of this type, and is used by our program as input.

We also must declare new variables for the stock list file, and for the items in the stock list:

```
var
   CustFile:      file of CustRec;
   Customer:      CustRec;
   StockFile:     file of ItemRec;
   Item:          ItemRec;
   Choice:        Char;

{$ISORT.BOX}
```

The last variable, *Choice*, is used in the main body of the program which lets us choose whether we want to sort the customer file or the stock file:

```
begin {program SortExampleOne}
   Write('Sort Customers or Stock? (enter C or S): ');
   repeat
      read(Kbd,Choice);
      Choice := UpCase(Choice);
   until Choice in ['C','S'];
   WriteLn(Choice);
   case Choice of
      'C': begin
              Assign(CustFile,'CUSTOMER.DTA');
              Reset(CustFile);
              WriteLn(TurboSort(SizeOf(CustRec)));
           end;
      'S': begin
              Assign(StockFile,'STOCK.DTA');
              Reset(StockFile);
              WriteLn(TurboSort(SizeOf(ItemRec)));
           end;
   end; {case}
end.
```

The program first prompts you to enter a **C** or an **S**, and then keeps reading the choice until one of these characters is entered. Note that the input is converted to upper-case, so your users can make their entries in either upperor lower-case.

Based on the choice the user makes, the **case** statement prepares the desired file for reading, and calls *TurboSort* while providing the size of the applicable data type as the parameter.

The *Inp* procedure also uses the variable *Choice* in a **case** statement to select the right file for reading:

```
procedure Inp;
begin
   case Choice of
      'C': begin
             repeat
               Read(CustFile,Customer);
               SortRelease(Customer);
             until EOF(CustFile);
           end;
      'S': begin
             repeat
               Read(StockFile,Item);
               SortRelease(Item);
             until EOF(StockFile);
           end;
   end; {case}
end;
```

If the choice is **C** for customers, the customer file is read, and its data passed on for sorting; if the choice is **S** for stock list, the stock list file is read.

In the *Less* function, we must declare two new variables of **type** *ItemRec*. Again, a **case** statement uses the variable *Choice* to determine which variables should be used in the comparison:

```
function Less;
var
   FirstCust:    CustRec absolute X;
   SecondCust:           CustRec absolute Y;
   FirstItem:    ItemRec absolute X;
   SecondItem:   ItemRec absolute Y;
begin
   case Choice of
      'C': Less := FirstCust.Number < SecondCust.Number;
      'S': Less := FirstItem.Price < SecondItem.Price;
   end;
end;
```

As you see, we use the field *Price* as the key for sorting the stock file.

The use of the **case** statement is repeated in the last procedure, *OutP*:

```
procedure OutP;
var
   I:Integer;
begin
   case Choice of
      'C': begin
             repeat
                SortReturn(Customer);
                with Customer do
                begin
                   Write(Number,' ',Name,' ');
                   for I := Length(Name) to 30 do Write(' ');
                   Write(Addr);
                   for I := Length(Addr) to 20 do Write(' ');
                   Write(City);
                   for I := Length(City) to 12 do Write(' ');
                   WriteLn(State,' ',Zip);
                end;
             until SortEOS;
          end;
      'S': begin
             repeat
                SortReturn(Item);
                with Item do
                begin
                   Write(Number,' ',Descrip,' ');
                   for I := Length(Descrip) to 30 do Write(' ');
                   WriteLn(InStock:5,Price:8:2);
                end;
             until SortEOS;
          end;
   end; {case}
end;
```

Keeping the actual processing in the **case** statements as we have done here is a good idea only as long as it is fairly simple to do. If your variousinput, output, and comparison routines become more complicated, it may be a good idea to isolate each in a separate subprogram, and then call these from the **case** statement, passing the necessary information as parameters.

## 10.2 Multiple Keys

Suppose you want to sort the stock data, not just on price as above, but on two keys: *primarily* on quantity in stock, and *secondarily* (if there are any of the specified items in stock) on price.

This is easy to do. Simply rewrite the comparison of *FirstItem* and *SecondItem* as follows:

```
function Less;
var
   FirstCust:          CustRec absolute X;
   SecondCust:         CustRec absolute Y;
   FirstItem:          ItemRec absolute X;
   SecondItem:         ItemRec absolute Y;
begin
   case Choice of
      'C': Less := FirstCust.Number < SecondCust.Number;
      'S': Less := (FirstItem.InStock < SecondItem.InStock) or
                   ((FirstItem.InStock = SecondItem.InStock) and
                    (FirstItem.Price < SecondItem.Price));
   end;
end;
```

You first compare the *InStock* fields. If one is larger than the other, this comparison determines which item is smaller. But if they are equal, the next comparison, between the *Price* fields, determines which data item is smaller.

You could carry this scheme further, and sort on as many fields as you desire.

## 10.3 Complete Example

The following is a listing of the complete example as described above. The file SORT2.PAS on your distribution diskette contains the example.

```
program SortExampleTwo { Customer File and Stock File };

type
   CustRec = record
                Number:        integer;
                Name:          string[30];
                Addr:          string[20];
                City:          string[12];
                State:         string[3];
                Zip:           string[5];
             end;
   ItemRec = record
                Number:        integer;
                Descrip:       string[30];
                InStock:       integer;
                Price:         real;
             end;
var
   CustFile:                   file of CustRec;
   Customer:                   CustRec;
   StockFile:                  file of ItemRec;
   Item:                       ItemRec;
   Choice:                     Char;
{$I SORT.BOX}
procedure Inp; {forward declared in SORT.BOX}
begin
   case Choice of
      'C': begin
              repeat
                 Read(CustFile,Customer);
                 SortRelease(Customer);
              until EOF(CustFile);
           end;
      'S': begin
              repeat
                 Read(StockFile,Item);
                 SortRelease(Item);
              until EOF(StockFile);
           end;
   end; {case}
end;
```

```
function Less; { boolean function with two parameters, }
                { X and Y, forward declared in SORT.BOX }
var
   FirstCust:                    CustRec absolute X;
   SecondCust:                   CustRec absolute Y;
   FirstItem:                    ItemRec absolute X;
   SecondItem:                   ItemRec absolute Y;
begin
   case Choice of
      'C': Less := FirstCust.Number < SecondCust.Number;
      'S': Less := (FirstItem.InStock < SecondItem.InStock) or
                   ((FirstItem.InStock = SecondItem.InStock) and
                    (FirstItem.Price < SecondItem.Price));
   end;
end;
procedure OutP; { forward declared in SORT.BOX }
var
   I:Integer;
begin
   case Choice of
      'C': begin
              repeat
                 SortReturn(Customer);
                 with Customer do
                 begin
                    Write(Number,' ',Name,' ');
                    for I := Length(Name) to 30 do Write(' ');
                    Write(Addr);
                    for I := Length(Addr) to 20 do Write(' ');
                    Write(City);
                    for I := Length(City) to 12 do Write(' ');
                    WriteLn(State,' ',Zip);
                 end;
              until SortEOS;
           end;
      'S': begin
              repeat
                 SortReturn(Item);
                 with Item do
                 begin
                    Write(Number,' ',Descrip,' ');
                    for I := Length(Descrip) to 30 do Write(' ');
                    WriteLn(InStock:5,Price:8:2);
                 end;
              until SortEOS;
           end;
```

```
    end; {case}
end;
begin {program SortExampleOne}
Write('Sort Customers or Stock? (enter C or S): ');
repeat
    read(Kbd,Choice);
    Choice := UpCase(Choice);
until Choice in ['C','S'];
WriteLn(Choice);
case Choice of
    'C': begin
            Assign(CustFile,'CUSTOMER.DTA');
            Reset(CustFile);
            WriteLn(TurboSort(SizeOf(CustRec)));
         end;
    'S': begin
            Assign(StockFile,'STOCK.DTA');
            Reset(StockFile);
            WriteLn(TurboSort(SizeOf(ItemRec)));
         end;
    end; {case}
    end.
```

# PART III

# GINST - General Installation System

# 11. INTRODUCTION TO GINST

This chapter explains how to use the GINST (General Installation) program to generate an installation routine for your TURBO Pascal programs.

## 11.1 This is GINST

This program creates installation programs which allow your customers to install your programs for their particular terminal(s). You may freely distribute the generated installation programs with any program you develop with TURBO Pascal.

## 11.2 Files on the Toolbox Diskette

The TURBO Toolbox distribution diskette contains a number of files corresponding to different TURBO tools. All tools have the extension .BOX. All sample programs are called .PAS. You should make a copy of the distribution diskette before you go to work to ensure that you always have the original, untouched versions of your TURBO tools.

The following files belong to the GINST system:

**GINST.COM** The GINST program. (.CMD in the CP/M-86 version)

**GINST.COD** Object code for the generated installation program. Must be present when you run GINST.

**GINST.MSG** Messages for GINST. These messages are also used to generate the .MSG file for your own installation program and must be present on the disk when you run GINST.

**GINST.DTA** Terminal installation data, used for generation the .DTA file for your own installation program. May be omitted if you create an installation program for an IBM PC.

**INSTALL.DOC** Documentation for the use of the installation program produced by GINST. You may include this text in your own manuals.

# NOTES

# 12. GENERATING AN INSTALLATION PROGRAM

GINST is not an installation program in itself; it is a program that **generates** installation programs which will then install TURBO Pascal programs.

When you start GINST, it first asks you to enter the name of the program which is to be installed by the generated installation program:

```
------------------------------------------------------------

                        Turbo Pascal
                Installation Program Generator


                        Version 2.00A
                Copyright© 1984 by Borland Inc.

------------------------------------------------------------

            Enter name of program to install: MYPROG
```

You can enter any legal file name; for example MYPROG. If you don't enter an extension, COM is assumed (CMD in the CP/M-86 version).

Next you must enter the **first** name you want to use for the generated installation program files, as one through eight characters:

```
Enter first name for installation files: MYINST
```

As an example, let's run the installation program for the TURBO Pascal program MYPROG to produce the installation program MYINST. GINST produces the following installation files:

```
Creating MYINST.COM
Creating MYINST.MSG
Creating MYINST.DTA

Installation program for MYPROG.COM created
```

That is all there is to generating an installation program. Refer to the *TURBO Pascal Reference Manual*: **Installation** for an explanation of how to install your program. The procedure is identical to installing TURBO Pascal itself.

**Note:** As a part of the conditions under which you purchased TURBO Toolbox, you may copy or paraphrase the **Installation** section ***only*** of the *TURBO Pascal Reference Manual* and include this information in your own documentation for your TURBO Pascal programs. This is the *only* extension of your privledges under the terms of the license agreements.

# A. ASCII TABLE

| DEC | HEX | CHAR | DEC | HEX | CHAR | DEC | HEX | CHAR | DEC | HEX | CHAR |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | ↑@ NUL | 32 | 20 | SPC | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 01 | ↑A SOH | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 02 | ↑B STX | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 03 | ↑C ETX | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 04 | ↑D EOT | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 05 | ↑E ENQ | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 06 | ↑F ACK | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 07 | ↑G BEL | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 08 | ↑H BS | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 09 | ↑I HT | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | ↑J LF | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | ↑K VT | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | ↑L FF | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | ↑M CR | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | ↑N SO | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | ↑O SI | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | ↑P DLE | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | ↑Q DC1 | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | ↑R DC2 | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | ↑S DC3 | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | ↑T DC4 | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | ↑U NAK | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | ↑V SYN | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | ↑W ETB | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | ↑X CAN | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | ↑Y EM | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | ↑Z SUB | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | ↑[ ESC | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | ↑\ FS | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | ↑] GS | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | ↑↑ RS | 62 | 3E | > | 94 | 5E | ↑ | 126 | 7E | ~ |
| 31 | 1F | ↑_ US | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | DEL |

# B. SUBJECT INDEX

## L



## M



## N



## O



## P

# V

# NOTES

# NOTES

NOTES

# NOTES

NOTES

## Something Totally New in Applications Software From Borland

# $49.95

**ALWAYS JUST A KEYSTROKE AWAY . . .**

# SIDEKICK™

## AVAILABLE FOR THE IBM PC, XT, jr. AND COMPATIBLES

**WHETHER YOU'RE RUNNING LOTUS 1-2-3, WORDSTAR, dBASEII OR WHATEVER . . .**

**JUST A KEYSTROKE AND A SIDEKICK WINDOW OPENS . . .**

- **A CALCULATOR**
- **A NOTEPAD**
- **AN APPOINTMENT CALENDAR**
- **AN AUTO DIALER**
- **A PHONE DIRECTORY**
- **AN ASCII TABLE**
- **AND MUCH MORE**

**ALL AT ONCE . . . OR ONE AT A TIME. ANYWHERE ON THE SCREEN YOU LIKE.**

**ANOTHER KEYSTROKE, AND YOU'RE RIGHT WHERE YOU LEFT OFF IN YOUR ORIGINAL PROGRAM!**
***(you never really left!)***

**Something brand new. Crafted in Assembly language as carefully as Borland's famous Turbo Pascal™, so that it's lightning-fast and as compact as only Borland knows how to make it!** With a notepad that has a full-screen editor that saves your notes to disk. You can even swap information back and forth between your applications software and your Sidekick™.

Suppose you're working with a spreadsheet, and you suddenly have an important idea. Just hit the button, a window opens, you write the note and hit the button again. You're right back where you left off in the spreadsheet.

Need to make a phone call? Whether the number is in an existing database, your own Sidekick phone directory, or you've just typed it on the screen . . . put the cursor next to the number, hit the keystroke, and Sidekick dials for you!*

There's lots more, too. You can move the Sidekick windows anywhere on the screen you like. And you can have as many on screen at a time as you need.

We designed it because we needed it. If you've ever been writing a report and needed to do a quick calculation, or jot down a note, then you understand why.

*Only with Hayes Smartmodem and compatibles.

**YOU CAN ORDER YOUR COPY OF SIDEKICK™ TODAY!**

*Just mail a check, money order or Visa or Mastercard number and expiration date.*

*Sidekick $49.95 + $5 shipping and handling. (California residents add 6% sales tax. Orders outside U.S., add $15 shipping and handling.)*

**Available directly from:**

BORLAND INTERNATIONAL

Borland International
4113 Sco
Scotts Va